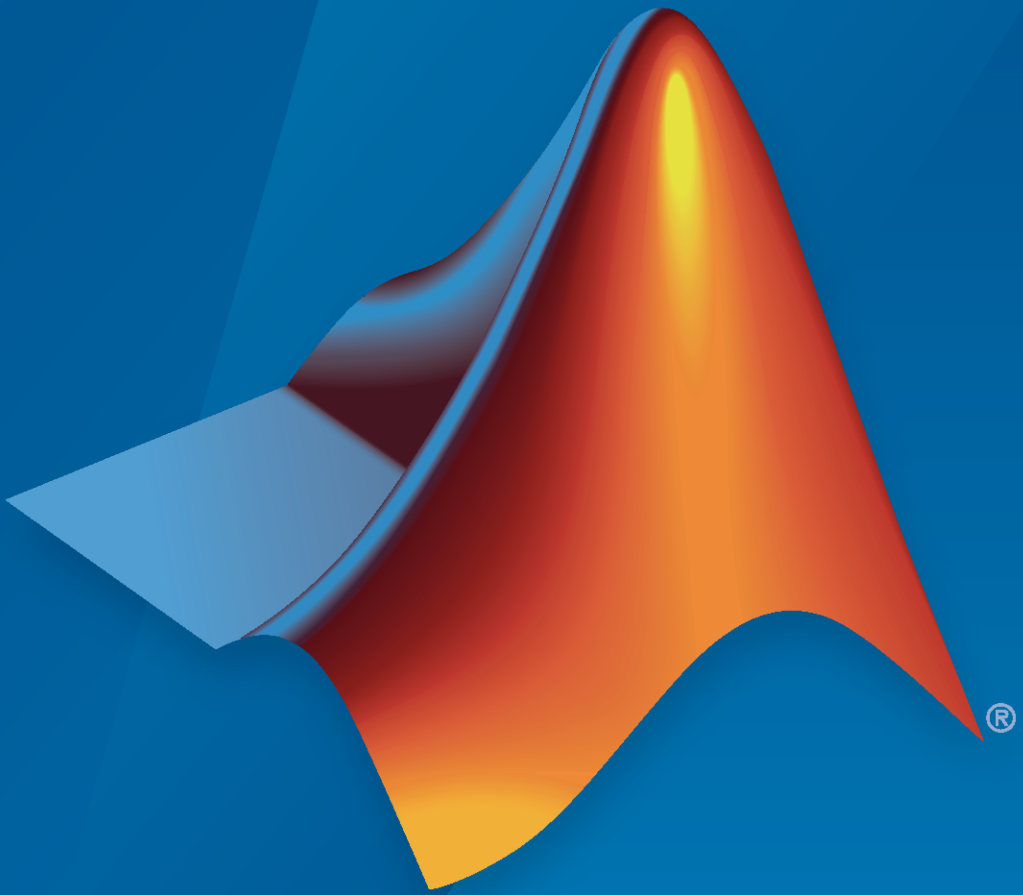
Industrial Communication Toolbox™

User's Guide



MATLAB® & SIMULINK®

R2023a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Revision History

June 2004	Online only	New for Version 1.0 (Release 14)
August 2004	Online only	Revised for Version 1.1 (Release 14+)
October 2004	Online only	Revised for Version 1.1.1 (Release 14SP1)
March 2005	Online only	Revised for Version 1.1.2 (Release 14SP2)
April 2005	Online only	Revised for Version 2.0 (Release 14SP2+)
September 2005	Online only	Revised for Version 2.0.1 (Release 14SP3)
March 2006	Online only	Revised for Version 2.0.2 (Release 2006a)
September 2006	Online only	Revised for Version 2.0.3 (Release 2006b)
March 2007	Online only	Revised for Version 2.0.4 (Release 2007a)
September 2007	Online only	Revised for Version 2.1 (Release 2007b)
March 2008	Online only	Revised for Version 2.1.1 (Release 2008a)
October 2008	Online only	Revised for Version 2.1.2 (Release 2008b)
March 2009	Online only	Revised for Version 2.1.3 (Release 2009a)
September 2009	Online only	Revised for Version 2.1.4 (Release 2009b)
March 2010	Online only	Revised for Version 2.1.5 (Release 2010a)
September 2010	Online only	Revised for Version 2.1.6 (Release 2010b)
April 2011	Online only	Revised for Version 3.0 (Release 2011a)
September 2011	Online only	Revised for Version 3.1 (Release 2011b)
March 2012	Online only	Revised for Version 3.1.1 (Release 2012a)
September 2012	Online only	Revised for Version 3.1.2 (Release 2012b)
March 2013	Online only	Revised for Version 3.2 (Release 2013a)
September 2013	Online only	Revised for Version 3.3 (Release 2013b)
March 2014	Online only	Revised for Version 3.3.1 (Release 2014a)
October 2014	Online only	Revised for Version 3.3.2 (Release 2014b)
March 2015	Online only	Revised for Version 3.3.3 (Release 2015a)
September 2015	Online only	Revised for Version 4.0 (Release 2015b)
March 2016	Online only	Revised for Version 4.0.1 (Release 2016a)
September 2016	Online only	Revised for Version 4.0.2 (Release 2016b)
March 2017	Online only	Revised for Version 4.0.3 (Release 2017a)
September 2017	Online only	Revised for Version 4.0.4 (Release 2017b)
March 2018	Online only	Revised for Version 4.0.5 (Release 2018a)
September 2018	Online only	Revised for Version 4.0.6 (Release 2018b)
March 2019	Online only	Revised for Version 4.0.7 (Release 2019a)
September 2019	Online only	Revised for Version 4.0.8 (Release 2019b)
March 2020	Online only	Revised for Version 5.0 (Release 2020a)
September 2020	Online only	Revised for Version 5.0.1 (Release 2020b)
March 2021	Online only	Revised for Version 5.0.2 (Release 2021a)
September 2021	Online only	Revised for Version 5.0.3 (Release 2021b)
March 2022	Online only	Revised for Version 6.0 (Release 2022a) (Renamed from *OPC Toolbox™ User's Guide*)
September 2022	Online only	Revised for Version 6.1 (Release 2022b)
March 2023	Online only	Revised for Version 6.2 (Release 2023a)

Contents

Getting Started

1 Introduction

Industrial Communication Toolbox Product Description **1-2**

Overview of OPC, Servers, and the Toolbox **1-3**
About Industrial Communication Toolbox Software **1-3**
About OPC **1-4**
OPC Servers **1-4**
System Requirements **1-5**

Get Command-Line Function Help **1-6**

Set Up Industrial Communication Toolbox Software for OPC **1-7**
Preparation Overview **1-7**
Set Up for Communicating with OPC DA and OPC HDA Servers **1-7**
Install an OPC DA or HDA Simulation Server for OPC Classic Examples **1-14**
Set Up for Communicating with OPC UA Servers **1-15**
Install an OPC UA Simulation Server for OPC UA Examples **1-15**
(Optional) Install a Local Discovery Service for OPC UA Server Discovery Examples **1-15**

Troubleshooting OPC Issues **1-17**
Unable to Find an OPC Server **1-17**
“Class not registered” Error **1-17**
Unable to Query the Server **1-17**
Unable to Connect to Server **1-17**
Unable to Create a Group **1-18**
Error While Querying Interface **1-18**

2 Quick Start: Using OPC Data Access Functions

Access Data at the Command Line **2-2**
DA Programming Overview **2-2**
Step 1: Locate Your OPC Data Access Server **2-2**
Step 2: Create an OPC Data Access Client Object **2-3**
Step 3: Connect to the OPC Data Access Server **2-3**

Step 4: Create an OPC Data Access Group Object **2-4**
Step 5: Browse the Server Name Space **2-4**
Step 6: Add OPC Data Access Items to the Group **2-5**
Step 7: View All Item Values **2-6**
Step 8: Configure Group Properties for Logging **2-6**
Step 9: Log OPC Server Data **2-7**
Step 10: Plot the Data **2-7**
Step 11: Clean Up **2-7**

3 Quick Start: Using the OPC Data Access Explorer

Access Data with the OPC Data Access Explorer **3-2**
Procedure Overview **3-2**
Step 1: Open the OPC Data Access Explorer **3-2**
Step 2: Locate Your OPC Server **3-3**
Step 3: Create an OPC Data Access Client Object **3-5**
Step 4: Connect to the OPC Server **3-7**
Step 5: Create an OPC Data Access Group Object **3-8**
Step 6: Browse the Server Name Space **3-10**
Step 7: Add OPC Data Access Items to the Group **3-12**
Step 8: View All Item Values **3-14**
Step 9: Configure Group Properties for Logging **3-15**
Step 10: Log OPC Server Data **3-17**
Step 11: Plot the Data **3-17**
Step 12: Clean Up **3-19**

4 Quick Start: Using OPC Historical Data Access Functions

Access Historical Data **4-2**
HDA Programming Overview **4-2**
Step 1: Locate Your OPC Historical Data Access Server **4-2**
Step 2: Create an OPC Historical Data Access Client Object **4-3**
Step 3: Connect to the OPC Historical Data Access Server **4-3**
Step 4: Retrieve Historical Data **4-3**
Step 5: Plot the Data **4-4**
Step 6: Clean Up **4-4**

Data Access User's Guide

5 Introduction to OPC Data Access (DA)

Discover Available Data Access Servers **5-2**
Prerequisites **5-2**
Determine Server IDs for a Host **5-2**

Connect to OPC Data Access Servers **5-4**
Overview **5-4**
Create a DA Client Object **5-4**
Connect a Client to the DA Server **5-5**
Browse the OPC DA Server Name Space **5-5**

6 Using OPC Data Access Objects

Create OPC Data Access Objects **6-2**
Overview to Objects **6-2**
Toolbox Object Hierarchy for the Data Access Standard **6-2**
How Toolbox OPC Objects Relate to OPC DA Servers **6-3**
Create Data Access Group Objects **6-4**
Create Data Access Item Objects **6-6**
Build an Object Hierarchy with a Disconnected Client **6-7**
Create OPC Data Access Object Vectors **6-8**
Work with Public Groups **6-10**

Configure OPC Data Access Object Properties **6-13**
Purpose of Object Properties **6-13**
View the Values of Object Properties **6-13**
View the Value of a Particular Property **6-14**
Get Information About Object Properties **6-15**
Set the Value of an Object Property **6-15**
View a List of All Settable Object Properties **6-16**

Delete Objects **6-18**

Save and Load Objects **6-20**

7 Reading, Writing, and Logging OPC Data

Read and Write Data on OPC DA Server **7-2**
Introduction to Reading and Writing **7-2**
Read Data from an Item **7-2**

Write Data to an Item **7-4**
Read and Write Multiple Values **7-6**

Data Change Events and Subscription **7-8**
Introduction to Data Change Events **7-8**
Configure OPC Objects for Data Change Events **7-8**
How Data Change Events are Processed **7-9**
Customize the Data Change Event Response **7-10**

Log OPC Server Data **7-11**
How Data Is Logged **7-11**
Configure a Logging Session **7-13**
Execute a Logging Task **7-15**
Get Logged Data into the MATLAB Workspace **7-16**

8 Working with OPC Data

OPC Data: Value, Quality, and TimeStamp **8-2**
Introduction to OPC Data **8-2**
Relationship Between Value, Quality, and TimeStamp **8-2**
How Value, Quality, and TimeStamp Are Obtained **8-3**

Work with Structure-Formatted Data **8-6**
When Structures Are Used **8-6**
Perform a Read Operation on Multiple Items **8-6**
Interpret Structure-Formatted Data **8-7**
When to Use Structure-Formatted Data **8-9**
Convert Structure-Formatted Data to Array Format **8-9**

Array-Formatted Data **8-11**
Array Content **8-11**
Conversion of Logged Data to Arrays **8-11**

Work with Different Data Types **8-13**
Conversion Between MATLAB Data Types and COM Variant Data Types **8-13**
Conversion of Values Written to an OPC Server **8-14**
Conversion of Values Read from an OPC Server **8-14**
Handling Arrays for Item Values **8-15**

9 Using Events and Callbacks

Use the Default Callback Function **9-2**
Overview to Callback Example **9-2**
Step 1: Create OPC Group Objects **9-2**
Step 2: Configure the Logging Task Properties **9-2**
Step 3: Configure the Callback Properties **9-2**

Step 4: Start the Logging Task **9-3**
Step 5: Clean Up **9-3**

Event Types **9-4**

Retrieve Event Information **9-8**
Event Structures **9-8**
Access Data in the Event Log **9-10**

Create and Execute Callback Functions **9-12**
Create Callback Functions **9-12**
Specify Callback Functions **9-13**
View Recently Logged Data **9-15**

10 Using the OPC Block Library

Block Library Overview **10-2**

Read and Write Data from a Model **10-3**
Example Overview **10-3**
Step 1: Create New Model in Simulink Editor **10-3**
Step 2: Open the OPC Block Library **10-3**
Step 3: Drag OPC Blocks into the Editor **10-4**
Step 4: Drag Other Blocks to Complete the Model **10-5**
Step 5: Configure OPC Servers for the Model **10-5**
Step 6: Specify the Block Parameter Values **10-7**
Step 7: Connect the Blocks **10-9**
Step 8: Run the Simulation **10-10**

Use the OPC Client Manager **10-11**
Introduction to the OPC Client Manager **10-11**
Add Clients to the OPC Client Manager **10-11**
Remove Clients from the OPC Client Manager **10-12**
Modify the Server Timeout Value for a Client **10-12**
Control Client/Server Connections **10-12**

11 Properties

Historical Data Access User's Guide

12 Introduction to OPC Historical Data Access (HDA)

OPC Historical Data Access **12-2**

Discover Available HDA Servers **12-4**
Prerequisites **12-4**
Determine HDA Server IDs for a Host **12-4**

Connect to OPC HDA Servers **12-5**
Overview **12-5**
Create an HDA Client Object **12-5**
View a Summary of a Client Object **12-5**
Connect an OPC HDA Client Object to the HDA Server **12-5**
Browse the OPC Server Name Space **12-6**
Get an OPC HDA Server Name Space **12-6**

13 Using OPC HDA Client Objects

OPC HDA Objects **13-2**

Locate an OPC HDA Server **13-3**

Create an OPC HDA Client Object **13-4**

Connect to the OPC HDA Server **13-5**
Browse the OPC Server Name Space **13-5**

Set Client Properties **13-6**
Set the Timeout Property **13-6**

Retrieve an OPC HDA Server Name Space **13-7**

Read Item Attributes **13-9**

14 Reading OPC Historical Data

Overview to Reading Historical Data **14-2**

Read Historical Data Over a Time Range **14-3**

Read Historical Data at Specific Times **14-4**

Read Processed Aggregate Data **14-5**

Retrieve Large Historical Data Sets **14-6**

Reading Modified Data **14-7**

Native MATLAB Data Types from Read Operations **14-8**
Request Structure Output **14-8**
Request MATLAB Numeric Data Output **14-8**
Request Cell Array Output **14-8**

Disconnect from HDA Servers **14-9**

Clean Up OPC HDA Objects **14-10**

15 Working with OPC HDA Data Objects

Introduction to OPC HDA Data Objects **15-2**

Display Data Objects **15-3**

OPC HDA Quality Values **15-4**

Manipulate Data Using OPC HDA Objects **15-5**
Resample Data Objects to Include All Available Time Stamps Using tsunion **15-5**
Resample Data Objects to Include All Common Time Stamps Using tsintersect **15-6**
Resample Data to a New Set of Time Stamps **15-6**
Convert OPC HDA Data Objects to MATLAB Numeric Data Types **15-7**

16 OPC HDA and UA Classes

Unified Architecture User's Guide

17 OPC Unified Architecture (UA)

About OPC Unified Architecture **17-2**

OPC UA Components **17-3**
Overview **17-3**
OPC UA Client **17-3**
OPC UA Node **17-3**
OPC UA Data **17-4**
OPC UA Quality **17-4**
Working with Time in OPC UA **17-4**

OPC UA Server Data Types **17-5**

OPC UA Security **17-7**

OPC UA Certificate Management **17-9**

OPC UA Aggregate Functions **17-10**
Introduction **17-10**
Available Aggregate Functions on an OPC UA Server **17-10**
OPC UA Standard Aggregate Functions **17-10**

Access Data from OPC UA Servers **17-13**
OPC UA Programming Overview **17-13**
Step 1: Locate Your OPC UA Server **17-13**
Step 2: Create an OPC UA Client and Connect to the Server **17-14**
Step 3: Browse OPC UA Server Namespace **17-15**
Step 4: Read Current Values from the OPC UA Server **17-16**
Step 5: Read Historical Data from the OPC UA Server **17-17**
Step 6: Plot the Data **17-18**
Step 7: Clean Up **17-18**

18 Controlling Devices Using Modbus

Modbus Interface Supported Features **18-2**
Modbus Capabilities **18-2**
Supported Platforms for Modbus **18-2**

Create a Modbus Connection **18-3**

Configure Properties for Modbus Communication **18-5**

Read Data from a Modbus Server **18-8**
Types of Data You Can Read over Modbus **18-8**
Read Coils over Modbus **18-8**
Read Inputs over Modbus **18-9**
Read Input Registers over Modbus **18-9**
Read Holding Registers over Modbus **18-10**
Specify Server ID and Precision **18-10**
Read Mixed Data Types **18-11**

Read Temperature from a Remote Temperature Sensor **18-13**

Write Data to a Modbus Server **18-14**
Types of Data You Can Write to over Modbus **18-14**
Write Coils over Modbus **18-14**
Write Holding Registers over Modbus **18-14**

Write and Read Multiple Holding Registers **18-16**

Modify the Contents of a Holding Register Using a Mask Write . . **18-18**

Use the Modbus Explorer App **18-19**

Configure a Connection in the Modbus Explorer **18-20**
Communicate over TCP/IP **18-20**
Communicate over Serial RTU **18-21**

Read Coils, Inputs, and Registers in the Modbus Explorer **18-23**
Edit the Read Registers Table **18-24**
Import or Export Read Data **18-25**

Write to Coils and Holding Registers in the Modbus Explorer . . . **18-26**

Control a PLC Using the Modbus Explorer **18-28**

Generate a Script from Your Modbus Explorer Session **18-33**

Troubleshooting the Modbus Interface **18-36**
Supported Platforms **18-36**

Configuration and Connection **18-36**
Other Troubleshooting Tips for Modbus **18-37**

OPC Information Reference

A OPC Quality

OPC Quality **A-2**

Major Quality **A-3**

Quality Substatus **A-4**

Limit Status **A-6**

B OPC DA Server Item Properties

OPC DA Server Item Properties **B-2**

OPC Item Property Set **B-3**

OPC Specific Properties **B-4**

OPC Recommended Properties **B-5**

C OPC HDA Item Attributes

OPC HDA Item Attributes **C-2**

19 Functions

20 Blocks

21 Industrial Communication Toolbox Examples

Install a Simulation Server for OPC Examples 21-2
Acquire Data from an OPC Data Access Server 21-3
Locate and Browse OPC Data Access Servers 21-7
Create and Configure OPC Objects 21-10
Manage OPC Data Access Objects 21-14
Read and Write Data to an OPC Data Access Server 21-17
Log Data from an OPC Data Access Server 21-21
View the OPC Event Log 21-24
Monitor Logging Progress with Callbacks 21-26
Update MATLAB Plots While Logging OPC Data 21-28
Locate and Browse OPC Historical Data Access Servers 21-30
Acquire Data from an OPC Historical Data Access Server 21-33
Visualize and Preprocess OPC HDA Data 21-37
Browse OPC UA Server Namespace 21-44
Read and Write Current OPC UA Server Data 21-51
Read Historical OPC UA Server Data 21-56
Visualize and Preprocess OPC UA Data 21-61
Read and Write to an OPC Data Access Server from Simulink 21-70
Use OPC Data to Test a Binary Distillation Column Plant Model 21-72

Get Started Accessing Data from a PI Server **21-74**

Read Data from a PI Server **21-77**

Process PI Data Using Common MATLAB Operations **21-81**

Get Started with MQTT **21-88**

Get Data from Subscribed Topics in an MQTT Client **21-90**

Subscribe to an MQTT Topic with a Callback Function **21-93**

Subscribe to an MQTT Wildcard Topic **21-95**

Getting Started

Introduction

- “Industrial Communication Toolbox Product Description” on page 1-2
- “Overview of OPC, Servers, and the Toolbox” on page 1-3
- “Get Command-Line Function Help” on page 1-6
- “Set Up Industrial Communication Toolbox Software for OPC” on page 1-7
- “Troubleshooting OPC Issues” on page 1-17

Industrial Communication Toolbox Product Description

Exchange data over OPC UA, Modbus, MQTT, and other industrial protocols

Industrial Communication Toolbox provides access to live and historical industrial plant data directly from MATLAB® and Simulink®. You can read, write, and log OPC Unified Architecture (UA) data from devices such as distributed control systems, supervisory control and data acquisition systems, and programmable logic controllers. You can also access plant and manufacturing data directly from OSIsoft® PI servers, and use this data for process monitoring, process improvement, and predictive maintenance applications.

You can work with data from live servers and data historians that conform to the OPC UA, OPC Data Access (DA), and OPC Classic Historical Data Access (HDA) standards. When communicating over OPC UA, you can securely connect to OPC UA servers using a variety of security modes, encryption algorithms, and user authentication methods.

The toolbox includes Simulink blocks that let you model online supervisory control and perform hardware-in-the-loop controller testing. In both MATLAB and Simulink, you can verify algorithms by establishing a secure OPC UA connection to your plant and build connected digital twin models for IIoT applications. The toolbox also supports communication with edge devices and cloud servers over Modbus® and MQTT protocols.

Overview of OPC, Servers, and the Toolbox

In this section...

"About Industrial Communication Toolbox Software" on page 1-3
"About OPC" on page 1-4
"OPC Servers" on page 1-4
"System Requirements" on page 1-5

About Industrial Communication Toolbox Software

Industrial Communication Toolbox software implements a hierarchical object-oriented approach to communicating with OPC servers using the OPC Data Access and Historical Data Access Standards. Using toolbox functions, you create OPC Data Access (DA) and Historical Data Access (HDA) Client objects which represent the connection between MATLAB and an OPC server. Using properties of the client objects you can control various aspects of the communication link, such as time out periods, connection status, and storage of events associated with that client. "Connect to OPC Data Access Servers" on page 5-4 and "Connect to OPC HDA Servers" on page 12-5 describe how to create DA and HDA client objects respectively.

Once you establish a connection to an OPC DA server, you create Data Access Group objects (`dagroup` objects) that represent collections of OPC Data Access Items. You then add Data Access Item objects (`daitem` objects) to that group, for monitoring server item values from the OPC server and writing values to the OPC server. You can use the `dagroup` object to perform such actions as determining how often the items in the group must be updated, executing a MATLAB function when the server provides notification of changes in item state, and other tasks related to the group. "Create OPC Data Access Objects" on page 6-2 describes how to create and configure `dagroup` objects and add `daitem` objects to a group.

Using OPC DA functionality, you can log records (a list of items that have changed, and their new values) from an OPC Data Access Server to disk or to memory, for later processing. The logging task is controlled by the `dagroup` object. "Log OPC Server Data" on page 7-11 describes how to log data using the OPC logging mechanism.

The HDA functionality allows for the retrieval and analysis of historical data from HDA OPC servers. Establishing a connection to an HDA server via the OPC HDA client object, allows you to retrieve historical data for a range of times or at a specific time. Both raw and aggregated data collections can be retrieved in the form of opc.hda.Data objects. These data objects provide numerous data manipulation and display operations.

To work with the data you acquire, you must bring it into the MATLAB workspace. When the records are acquired, the toolbox stores them in a memory buffer or on disk. The toolbox provides several ways to bring one or more records of data into the workspace where you can analyze or visualize the data.

You can enhance your OPC application by using DA event callbacks. The toolbox has defined certain OPC software occurrences, such as the start of an acquisition task, as well as OPC server initiated occurrences, such as notification that an item's state has changed, as events. You can associate the execution of a particular function with a particular event.

When working in the Simulink environment, you can use blocks from the OPC block library to use live OPC data as inputs to your model and update the OPC server with your model outputs. The Industrial

Communication Toolbox OPC block library includes the capability of running Simulink models in pseudo real time, by slowing the simulation to match the system clock. You can prototype control systems, provide plant simulators, and perform optimization and tuning tasks using Simulink and the Industrial Communication Toolbox OPC block library.

About OPC

Open Platform Communications (OPC) is a set of interoperability standards maintained by the OPC Foundation (`https://opcfoundation.org`) for the exchange of data in the industrial automation and other industries. OPC uses Microsoft® DCOM technology to provide a communication link between OPC servers and OPC clients. OPC has been designed to provide reliable communication of information in a process plant, such as a petrochemical refinery, an automobile assembly line, or a paper mill.

Before you interact with OPC servers using Industrial Communication Toolbox software, you should understand the OPC client-server relationship, how OPC servers organize their server items, and how clients can interact with those server items. “Toolbox Object Hierarchy for the Data Access Standard” on page 6-2 explains these concepts in detail.

OPC Servers

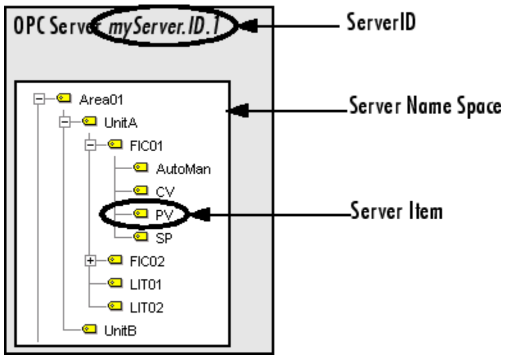
Industrial Communication Toolbox software acts an OPC Data Access and Historical Data Access *client* application, capable of connecting to any OPC DA and HDA compliant *server*. By utilizing the OPC Foundation standards, the toolbox does not require any knowledge about the internal configuration and operation of the OPC server. Instead, the OPC Standard provides the common mechanism for the server and client to interact with each other.

An OPC server is identified by a unique server ID. The server ID is unique to the computer on which the server is located. A combination of the host name of the server computer, and the server ID of the OPC server, provides a unique identifier for an OPC server on a network of computers.

OPC Server Name Spaces

All OPC servers are required to publish a name space, consisting of an arrangement of the name of every server item (also known as an item ID) associated with that server. The name space provides the internal map of every device and location that the server is able to monitor and/or update.

The following figure shows a portion of the name space on a typical OPC server.



Server Item

A *server item* represents a value on the OPC server that a client might be interested in. A server item could represent a physical measurement device (such as a temperature sensor), a particular component of a device (such as the set-point for a controller), or a variable or storage location in a supervisory control and data acquisition (SCADA) system. Each server item is uniquely represented on the server by a fully qualified item ID. The fully qualified item ID is usually made up of the path to that server item in the tree, with each node name separated by a period character. In the previous Server Item figure, the fully qualified item ID for the highlighted server item might be `Area01.UnitA.FIC01.PV`.

Most OPC servers provide a hierarchical name space, where server items are arranged in a tree-like structure. The tree can contain many different categories (called *branch nodes*), each with one or more branches and/or *leaf nodes*. A leaf node contains no other branches, and often represents a specific server Item. The fully qualified item ID of a server item is simply the `path' to that leaf node, with a server-dependent separator.

Some OPC servers provide only a flat name space, where server items are all arranged in one single group. You could consider a flat name space as a name space containing only leaf nodes.

It is possible to convert a hierarchical name space into a flat name space. It is not always possible to convert a flat name space into a hierarchical name space.

For information on how to obtain the name space of an OPC server, see “Browse the OPC Server Name Space” on page 12-6.

System Requirements

Industrial Communication Toolbox software provides the OPC Data Access client capabilities from within MATLAB. To use this toolbox functionality, you need access to an OPC server that supports the Data Access Specification version 2.05. In addition, you will need to ensure that you are able to connect to those OPC servers from the computer on which the toolbox software is installed. For more information on how to configure the client and server computers so that you can connect to an OPC server, see “Set Up Industrial Communication Toolbox Software for OPC” on page 1-7.

Get Command-Line Function Help

To get command-line function help, use the MATLAB `help` function. For example, to get help for the `opcserverinfo` function, type

```
help opcserverinfo
```

To get help on a particular OPC HDA function, use the `opchda` prefix. For example to get help on the HDA equivalent of the `opcserverinfo` function, type

```
help opchdaserverinfo
```

Industrial Communication Toolbox software also provides its own versions of several MATLAB functions, using the same function names. For example, the toolbox provides a version of the `isvalid` function. When you type

```
help isvalid
```

you get help for the MATLAB handle object version of this function. If there are multiple versions of a function available, the help indicates this. For `isvalid`, the help contains this line:

```
Other functions named isvalid
```

If necessary, click that link to view the function list. You might see a listing like this.

```
Other functions named isvalid:
       handle/isvalid, timer/isvalid, serial/isvalid, instrument/isvalid,
       imaqdevice/isvalid, imaqchild/isvalid, vrworld/isvalid,
       vrnode/isvalid, vrfigure/isvalid, daqdevice/isvalid,
       daqchild/isvalid, icgroup/isvalid, xregpointer/isvalid,
       idnlgrey/isvalid, iconnect/isvalid, opcroot/isvalid.
```

To get help on the Industrial Communication Toolbox version of this function, click the appropriate link, or type

```
help opcroot/isvalid
```

To avoid specifying which version to view, use the `opchelp` function.

```
opchelp isvalid
```

You can also use `opchelp` to get help on OPC object properties.

```
opchelp EventLog
```

Set Up Industrial Communication Toolbox Software for OPC

In this section...

"Preparation Overview" on page 1-7

"Set Up for Communicating with OPC DA and OPC HDA Servers" on page 1-7

"Install an OPC DA or HDA Simulation Server for OPC Classic Examples" on page 1-14

"Set Up for Communicating with OPC UA Servers" on page 1-15

"Install an OPC UA Simulation Server for OPC UA Examples" on page 1-15

"(Optional) Install a Local Discovery Service for OPC UA Server Discovery Examples" on page 1-15

Preparation Overview

Before you can communicate with OPC servers on your network, you need to prepare your workstation (and possibly the OPC server host computer) to use the technologies on which Industrial Communication Toolbox software is built. These technologies, described in "About OPC" on page 1-4, allow you to browse for and connect to OPC servers on your network, and allow those OPC servers to interact with your MATLAB session using Industrial Communication Toolbox OPC software.

The specific steps are described in the following sections.

Set Up for Communicating with OPC DA and OPC HDA Servers

Install the OPC Foundation Core Components

OPC DA and HDA use the "OPC Classic" technologies, which employ Microsoft DCOM standards. DCOM is used for client-server communication, and for managing security of the connections through standard Microsoft security permissions on DCOM objects. To use OPC Classic capabilities, you must configure your computer and possibly the server computer to allow for this communication.

The OPC Foundation provides a set of tools for browsing other computers on your network for OPC servers, and for communicating with the OPC servers. These tools are called the OPC Foundation Core Components, and are shipped with Industrial Communication Toolbox software.

To install the OPC Foundation Core Components, use the `opcregister` function. You can also use the `opcregister` function to remove or repair the OPC Foundation Core Components installation.

Installing, repairing, and removing the OPC Foundation Core Components follows the same steps:

1. If you are repairing or removing the OPC Foundation Core Components, make sure that you do not have any OPC objects in memory. Use the `opcreset` function to clear all objects from memory.

   ```
   opcreset;
   ```

2. Run `opcregister` with the action you would like to perform. If you do not supply an option, the function assumes that you want to *install* the components. Otherwise, use `'repair'` to *repair* an installation (reinstall the files), or `'remove'` to *remove* the components.

   ```
   opcregister('install')
   ```

3. You will be prompted to type `Yes` to confirm the action you want to perform. You must type `Yes` exactly as shown, without any quotes. This confirmation question is used to ensure that you acknowledge the action that is about to take place.
4. The OPC Foundation Core Components will be installed, repaired, or removed from your system.
5. If you receive a warning about having to reboot your computer, you must quit MATLAB and restart your computer for the changes to take effect.

Configure DCOM

DCOM is a client-server based architecture for enabling communication between two applications running on distributed computers. The OPC DA and HDA specifications utilize DCOM for communication between the OPC client (for example, Industrial Communication Toolbox software) and the OPC server. To successfully use DCOM, those two computers must share a common security configuration so that the two applications are granted the necessary rights to communicate with each other.

To connect successfully to OPC Servers using Industrial Communication Toolbox, you must configure DCOM permissions between the client computer (on which MATLAB is installed) and the server computer (running the OPC Server). This section describes two typical DCOM configuration options for Industrial Communication Toolbox software. Other DCOM options might provide sufficient permissions for the toolbox to work with an OPC server; the options described here are known to work with tested vendors' OPC servers.

There are two configuration types described in this section:

- "Configure DCOM to Use Named User Security" on page 1-9 describes how to provide security between the client and server negotiated on a dedicated named user basis. You do not have to be logged in as the named user in order to use this mechanism; all communications between the client and the server are performed using the dedicated named user, independently of the user making the OPC requests. However, the identity used to run the OPC server must be available on the client machine, and the password of that identity must match on both machines.
- "Configure DCOM to Use No Security" on page 1-13 describes a configuration that provides no security between the client and server. Use this option only if you are connecting to an OPC server on a dedicated, private network. This configuration option has been known to cause some Microsoft Windows® services to fail, and to leave the computer vulnerable to malicious intrusion from other network users.

You should use the named user configuration, unless your system administrator indicates that no security is required for OPC access.

Caution If your OPC server software comes with DCOM setup guidelines, you should first attempt to follow the instructions provided by the OPC server vendor. The guidelines provided in this section are generic and may not suit your specific network and security model.

Note The following instructions apply to the Microsoft Windows 7 operating system with Service Pack 1. Users of other Microsoft Windows operating systems should be able to adapt these instructions to configure DCOM on their systems.

Configure DCOM to Use Named User Security

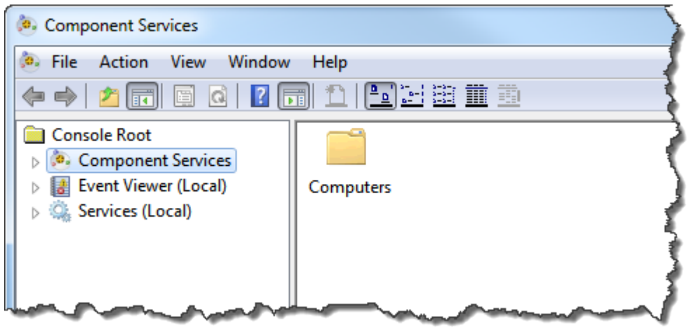
To configure DCOM to use named user security, you will have to ensure that both the server machine and client machine have a common user who is granted DCOM access rights on both the server and client machines. You should consult the following sections for information on configuring each machine:

- “OPC Server Machine Configuration” on page 1-9 provides the steps that you must perform on each of the machines providing OPC servers.
- “Client Machine Configuration” on page 1-10 provides the steps that you must perform on the machine that will run MATLAB and Industrial Communication Toolbox software.

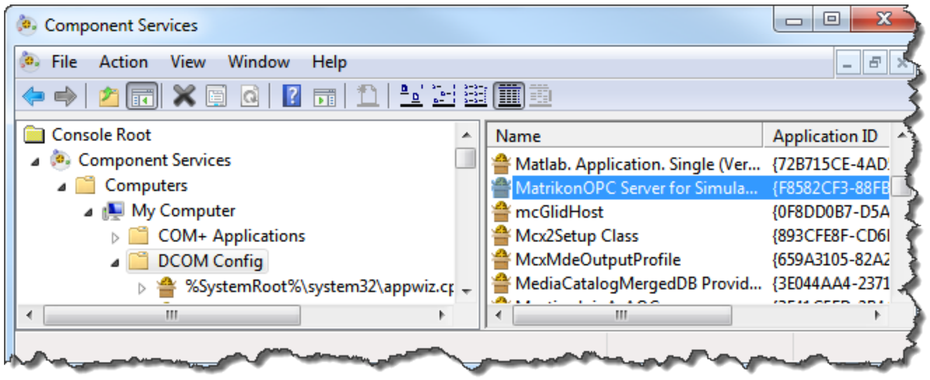
OPC Server Machine Configuration

On the machines hosting the OPC servers, perform the following steps:

1. Create a new local user. (You can also create a domain user if the server and client machines are part of the same domain.) The name used in these instructions is `opc` (displayed as `OPC Server` in dialogs boxes), but you can choose any name, as long as you remain consistent throughout these instructions.
2. Select **Start > Control Panel**. Double-click `Administrative Tools` and then double-click `Component Services`. The Component Services dialog appears.



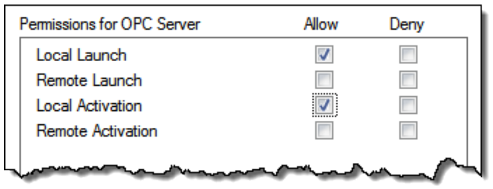
3. Browse to `Component Services > Computers > My Computer > DCOM Config`.
4. Locate your OPC server in the `DCOM Config` list. The example below shows the Matrikon™ OPC Server for Simulation.



Name	Application ID
Matlab. Application. Single (Ver...	{72B715CE-4AD
MatrikonOPC Server for Simula...	{F8582CF3-88FB
mcGlidHost	{0F8DD0B7-D5A
Mcx2Setup Class	{893CFE8F-CD6I
McxMdeOutputProfile	{659A3105-82A2
MediaCatalogMergedDB Provid...	{3E044AA4-2371

5. Right-click the OPC server object, and choose **Properties**.
6. In the **General** tab, ensure that the Authentication Level is set to `Default` or to `Connect`.

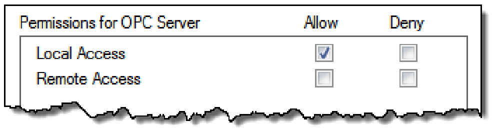
7 In the **Security** tab, choose Customize for the Launch and Activation Permissions, then click **Edit**. Ensure that the `opc` user is granted local Launch and Activation permissions.



Permissions for OPC Server	Allow	Deny
Local Launch	✓	
Remote Launch		
Local Activation	✓	
Remote Activation		

Click **OK** to dismiss the Local Launch and Activation Permissions dialog box.

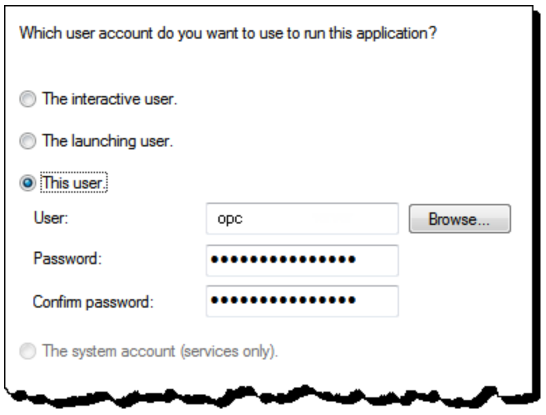
8 In the **Security** tab, choose Customize for the Access Permissions, then click **Edit**. Ensure that the `opc` user is granted Local Access permissions.



Permissions for OPC Server	Allow	Deny
Local Access	✓	
Remote Access		

Click **OK** to dismiss the Local Launch and Activation Permissions dialog box.

9 In the **Identity** tab, select **This user** and type the name and password for the `opc` user (created in step 1).



Which user account do you want to use to run this application?

The interactive user.

The launching user.

[✓ This user.]

User: opc

Password: ••••••••••••••••

Confirm password: ••••••••••••••••

The system account (services only).

10 If the OPC server runs as a service, make sure that the service runs as the `opc` user (created in step 1) and not as the system account. Consult your system administrator for information on how to configure a service to run as a specific user.

11 Repeat steps 4 through 10 for each of the servers you want to connect to.

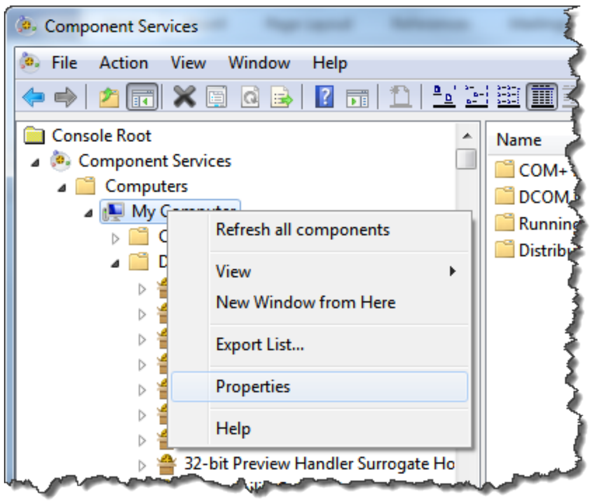
Client Machine Configuration

On the machine(s) that will be running MATLAB and Industrial Communication Toolbox software, perform the following steps:

1 On the client machine(s), create the identical local user with the same name and password permissions as you set up in step 1 of “OPC Server Machine Configuration” on page 1-9.

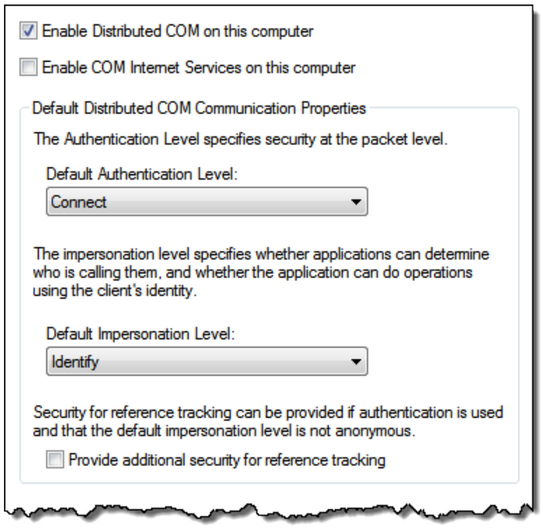
2 Select **Start > Control Panel**. Double-click `Administrative Tools` and then double-click `Component Services`. The Component Services dialog appears.

3 Browse to `Component Services > Computers > My Computer`. Right-click `My Computer` and select **Properties**.



4 Click the **Default Properties** tab, and ensure that:

- Enable Distributed COM is checked
- Default Authentication Level is set to `Connect`
- Default Impersonation Level is set to `Identify`



[✓] Enable Distributed COM on this computer

[ ] Enable COM Internet Services on this computer

Default Distributed COM Communication Properties

The Authentication Level specifies security at the packet level.

Default Authentication Level:

Connect

The impersonation level specifies whether applications can determine who is calling them, and whether the application can do operations using the client's identity.

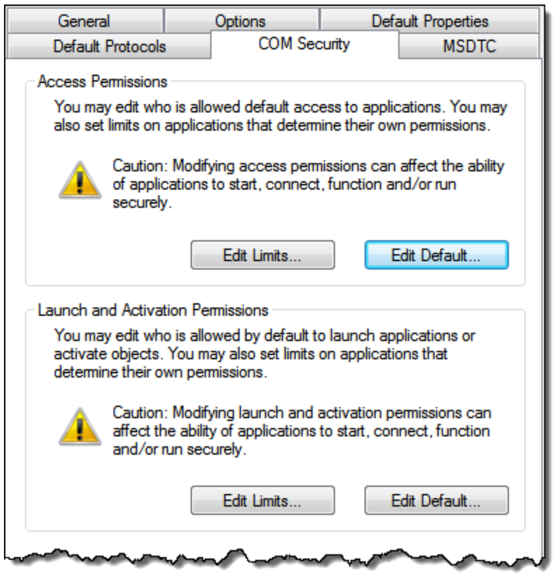
Default Impersonation Level:

Identify

Security for reference tracking can be provided if authentication is used and that the default impersonation level is not anonymous.

[ ] Provide additional security for reference tracking

5 Click the **COM Security** tab.



Access Permissions

You may edit who is allowed default access to applications. You may also set limits on applications that determine their own permissions.

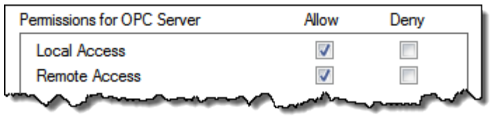
Caution: Modifying access permissions can affect the ability of applications to start, connect, function and/or run securely.

Launch and Activation Permissions

You may edit who is allowed by default to launch applications or activate objects. You may also set limits on applications that determine their own permissions.

Caution: Modifying launch and activation permissions can affect the ability of applications to start, connect, function and/or run securely.

6 For the Access Permissions, click **Edit Default** and ensure that the `opc` user is included in the Default Security list, and is granted both Local Access and Remote Access permissions.



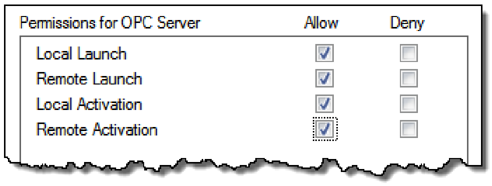
Permissions for OPC Server	Allow	Deny
Local Access	✓	
Remote Access	✓	

Click **OK** to close the Default Access Permissions dialog box.

7 Still under Access Permission", click **Edit Limits** and ensure that the `opc` user is included in the Security Limits list, and is granted both Local Access and Remote Access permissions.

Click **OK** to close the Security Limits dialog box.

8 For the Launch and Activation permissions, click **Edit Default** and ensure that the `opc` user is included in the Default Security list, and is granted all rights (Local Launch, Remote Launch, Local Activation, and Remote Activation).



Permissions for OPC Server	Allow	Deny
Local Launch	✓	
Remote Launch	✓	
Local Activation	✓	
Remote Activation	✓	

Click **OK** to close the Default Access Permissions dialog box.

9 Still under Launch and Activation Permission, click **Edit Limits** and ensure that the `opc` user is included in the Security Limits list, and is granted all rights (Local Launch, Remote Launch, Local Activation, and Remote Activation).

Click **OK** to close the Security Limits dialog.

10 Click **OK**. A dialog warns you that you are modifying machine-wide DCOM settings.

Click **Yes** to accept the changes.

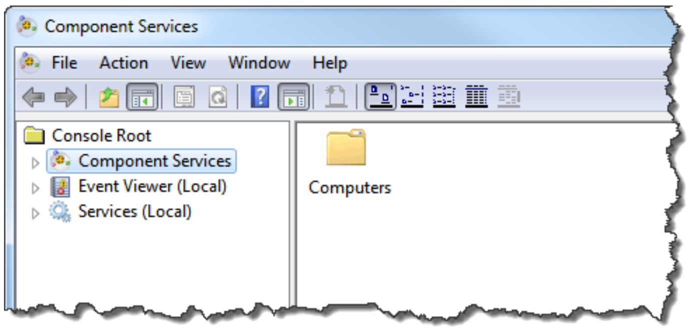
Your local client machine and server applications are now configured to use the same username when the server attempts to establish a connection back to the client.

Configure DCOM to Use No Security

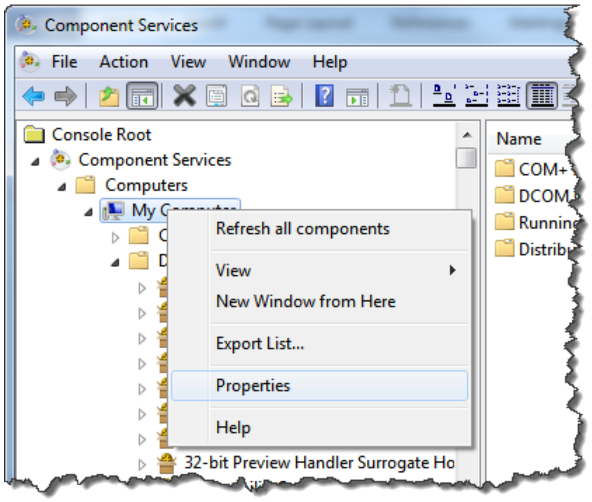
Caution You should not use this option if you are not in a completely trusted network. Turning off DCOM security means that any user on the network can launch any COM object on your local machine. Consult your network administrator before following these instructions.

You must complete the following steps on *both* the client and server machines.

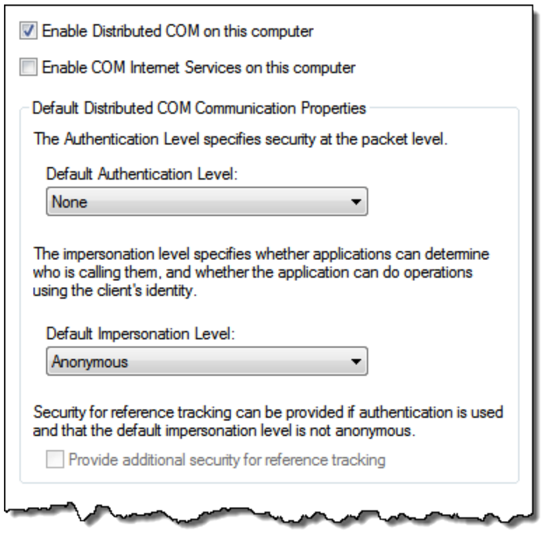
1. Ensure that the `Guest` user account is enabled. (The `Guest` account is disabled by default on Windows 7 machines). Consult your system administrator for information on how to enable the `Guest` account.
2. Select **Start > Control Panel**. Double-click `Administrative Tools` and then double-click `Component Services`. The Component Services dialog appears.



3. Browse to `Component Services > Computers > My Computer`. Right-click `My Computer` and select **Properties**.



4. In the **Default Properties** tab, make sure that Enable Distributed COM On This Computer is selected. Select None as the Default Authentication Level, and Anonymous as the Default Impersonation Level.


[✓] Enable Distributed COM on this computer
[ ] Enable COM Internet Services on this computer

Default Distributed COM Communication Properties

The Authentication Level specifies security at the packet level.

Default Authentication Level: None

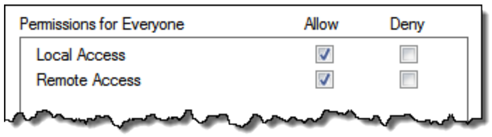
The impersonation level specifies whether applications can determine who is calling them, and whether the application can do operations using the client's identity.

Default Impersonation Level: Anonymous

Security for reference tracking can be provided if authentication is used and that the default impersonation level is not anonymous.

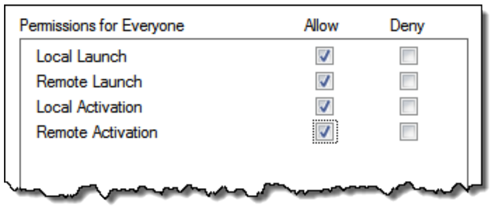
[ ] Provide additional security for reference tracking

5 In the COM Security tab, select Edit Limits from the Access Permissions and ensure that `Everyone` and `ANONYMOUS LOGON` are both granted Local Access and Remote Access.



Permissions for Everyone	Allow	Deny
Local Access	✓	
Remote Access	✓	

6 In the COM Security tab, select Edit Limits from the Launch and Activation Permissions and ensure that `Everyone` and `ANONYMOUS LOGON` are both granted Local and Remote permissions (Local Launch, Remote Launch, Local Activation and Remote Activation).



Permissions for Everyone	Allow	Deny
Local Launch	✓	
Remote Launch	✓	
Local Activation	✓	
Remote Activation	✓	

Both the client and the server are now configured so that anybody can access any COM object on either machine.

Caution This configuration is potentially dangerous in terms of security, and is recommended for debugging purposes only.

Install an OPC DA or HDA Simulation Server for OPC Classic Examples

OPC DA and OPC HDA (together, called "OPC Classic") examples in this guide and in the Industrial Communication Toolbox online help make use of a Matrikon OPC Simulation Server that you can download free of charge from https://www.matrikonopc.com.

Note You do not need to install the Matrikon OPC Simulation Server to enable the OPC functionality of Industrial Communication Toolbox. The Simulation Server is used here only for showing examples of the capabilities and syntax of OPC commands, and for providing fully working examples.

To install the Matrikon OPC Simulation Server, follow the installation instructions with the software. The Industrial Communication Toolbox documentation and OPC examples assume a default installation of the Matrikon Simulation Server.

Set Up for Communicating with OPC UA Servers

Allow OPC UA Communication Through Firewalls

OPC UA communication takes place using various TCP/IP ports. To locate OPC UA servers on other hosts, Industrial Communication Toolbox uses the OPC UA Local Discovery Service for that host, which is hosted on port 4840. Every other OPC UA server on a host uses a different port for communication. Locally, Industrial Communication Toolbox uses a random local port number to initiate the connection.

If you have a local firewall, you must ensure that the firewall allows MATLAB to communicate through the firewall. All other firewalls between the Industrial Communication Toolbox software and the OPC UA servers must permit communication on port 4840 plus all other ports set up by your OPC server administrator for the OPC UA servers you want to connect to.

Install an OPC UA Simulation Server for OPC UA Examples

OPC UA examples in this documentation make use of a Prosys OPC UA Simulation Server that you can download free of charge from https://www.prosysopc.com/products/opc-ua-simulation-server/.

To install the Prosys OPC UA Simulation Server, follow the installation instructions with the software. When you have started the server, you might want to reduce the number of ports used by the server by turning off HTTPS endpoints in the **Endpoints** tab of the Prosys OPC UA Simulation Server tool.

(Optional) Install a Local Discovery Service for OPC UA Server Discovery Examples

If you want to explore the OPC UA server discovery examples, you must install the OPC UA Local Discovery Service (LDS) and register your Simulation Server with the LDS. A free LDS installer is maintained by the OPC Foundation.

Download the Local Discovery Service

Download the LDS installer from https://opcfoundation.org/developer-tools/samples-and-tools-unified-architecture/local-discovery-server-lds/

The download is free, although you must create an OPC Foundation website account to access downloads.

Run the installer, which automatically registers the LDS on your computer. The LDS always uses port 4840 for communication.

Register the Simulation Server with the Local Discovery Service

The LDS requires a secure connection to OPC UA servers to allow those servers to register successfully with the LDS. This requires an Application Instance certificate to be trusted by the LDS. To allow the Prosys OPC UA Simulation Server to register with the OPC Foundation LDS, follow these steps.

1. Run the Prosys OPC UA Simulation Server.
2. Select **Options** > **Switch to Expert Mode**.
3. In the **Endpoints** tab, **Register to** pane, check the option `Local Discovery Server`.
4. In the **Certificates** tab, select the `SimulationServer` node and click **Open in File Explorer**.
5. Copy all files in the folder to `C:\ProgramData\OPC Foundation\UA\Discovery\pki\trusted\certs`.
6. Restart the Prosys OPC UA Simulation Server.
7. In the MATLAB Command Window, discover OPC UA servers published by the LDS. You should see an entry named `SimulationServer`.

   ```
   s = opcuaserverinfo('localhost')
   ```

   ```
   s =

   OPC UA ServerInfo 'SimulationServer':

      Connection Information
       Hostname: 'opc-demo1.my.local'
           Port: 53530
   ```
8. Create an OPC UA client, and connect in to the simulation server in MATLAB:

   ```
   opcua(s);
   connect(s)
   ```

   Depending on the server configuration, you might see an error on your initial attempt to connect:

   ```
   Error using opc.ua.Client/Connect
   An error occurred verifying security
   ```

   To correct this, you must manually mark the certificate as trusted on the server side:

   a. Open the Prosys OPC UA Simulation Server tool.
   b. Select **Options** > **Switch to Expert Mode**.
   c. In the **Certificates** tab, right-click the `MATLAB OPC Toolbox` entry, and select **Trusted**. Now you can connect.

Troubleshooting OPC Issues

In this section...

"Unable to Find an OPC Server" on page 1-17

""Class not registered" Error" on page 1-17

"Unable to Query the Server" on page 1-17

"Unable to Connect to Server" on page 1-17

"Unable to Create a Group" on page 1-18

"Error While Querying Interface" on page 1-18

If you are unable to establish a connection to an OPC server, the following sections might help you to identify and solve problems with installation and configuration that could be preventing you from successfully querying and connecting to OPC servers.

Most problems with connecting to an OPC server relate to the DCOM settings on either the host or the client machine. For information on configuring DCOM, see "Configure DCOM" on page 1-8.

Unable to Find an OPC Server

First, check that you are able to communicate with the host from your client. You can test this by attempting to run a Command Prompt and using the `'ping'` command on the host. Alternatively, try to browse to the host using the Network Neighborhood.

If you are able to communicate with the host, but you are unable to find an OPC server (using the `opcserverinfo` command) on that host, then the OPC Foundation Core Components may have to be reinstalled on your workstation. You can run the `opcregister` function to repair your OPC Foundation Core Components installation. For more information see "Install the OPC Foundation Core Components" on page 1-7.

"Class not registered" Error

If you get this error while attempting to query a server using `opcserverinfo`, or when attempting to add a host in the **OPC Data Access Explorer** app, the OPC Foundation Core Components have not been installed correctly. Install the OPC Foundation Core Components, as described in "Install the OPC Foundation Core Components" on page 1-7.

Unable to Query the Server

If you are unable to query the server using `opcserverinfo`, the most common cause is incorrectly configured local DCOM security settings. Review the section on "Configure DCOM" on page 1-8.

Unable to Connect to Server

An inability to connect to the OPC server usually indicates that the security model on the server is not allowing you to make an initial connection. Check the DCOM configuration on the server, and review the section on "Configure DCOM" on page 1-8.

Unable to Create a Group

If you are able to connect to the server but cannot create a group, the most common cause is incorrectly configured local DCOM security settings. Review the section on “Configure DCOM” on page 1-8.

Error While Querying Interface

If you get this error while attempting to add a group to a connected client object,

```
Error occurred while querying interface: IID_IOPCDataCallback
```

your local DCOM security settings are not permitting the OPC server to connect to the Industrial Communication Toolbox OPC client on the local machine. Review the section on “Configure DCOM” on page 1-8.

Quick Start: Using OPC Data Access Functions

The best way to learn about Industrial Communication Toolbox OPC capabilities is to look at a simple example. This topic illustrates the basic steps required to log data from an OPC Data Access (DA) server for analysis and visualization.

Cross-references to other sections in the documentation provide more in-depth discussions of the relevant concepts.

Access Data at the Command Line

In this section...

“DA Programming Overview” on page 2-2
“Step 1: Locate Your OPC Data Access Server” on page 2-2
“Step 2: Create an OPC Data Access Client Object” on page 2-3
“Step 3: Connect to the OPC Data Access Server” on page 2-3
“Step 4: Create an OPC Data Access Group Object” on page 2-4
“Step 5: Browse the Server Name Space” on page 2-4
“Step 6: Add OPC Data Access Items to the Group” on page 2-5
“Step 7: View All Item Values” on page 2-6
“Step 8: Configure Group Properties for Logging” on page 2-6
“Step 9: Log OPC Server Data” on page 2-7
“Step 10: Plot the Data” on page 2-7
“Step 11: Clean Up” on page 2-7

DA Programming Overview

This section illustrates the basic steps to create an OPC Data Access application by visualizing the Triangle Wave and Saw-toothed Wave signals provided by the Matrikon OPC Simulation Server. The application logs data to memory and plots that data, highlighting uncertain or bad data points. By visualizing the data you can more clearly see the relationships between the signals.

Note To run the sample code in the following steps you need the Matrikon OPC Simulation Server on your local machine. For installation details, see “Install an OPC DA or HDA Simulation Server for OPC Classic Examples” on page 1-14. The code requires only minor changes to work with other servers.

Step 1: Locate Your OPC Data Access Server

In this step, you obtain two pieces of information that the toolbox needs to uniquely identify the OPC Data Access server that you want to connect to. You use this information when creating an OPC Data Access Client object (`opcda` client object), described in “Step 2: Create an OPC Data Access Client Object” on page 2-3.

The first piece of information is the host name of the server computer. The host name (a descriptive name like "`PlantServer`" or an IP address such as `192.168.16.32`) qualifies that computer on the network, and is used by the OPC Data Access protocols to determine the available OPC servers on that computer, and to communicate with the computer to establish a connection to the server. In any OPC application, you must know the name of the OPC server's host, so that a connection with that host can be established. Your network administrator can provide a list of host names that provide OPC servers on your network. In this example, you will use `localhost` as the host name, because you will connect to the OPC server on the same machine as the client.

The second piece of information is the OPC server's *server ID*. Each OPC server on a particular host is identified by a unique server ID (also called the Program ID or ProgID), which is allocated to that server on installation. The server ID is a text character vector, usually containing periods.

Although your network administrator can provide a list of server IDs for a particular host, you can query the host for all available OPC servers. “Discover Available Data Access Servers” on page 5-2 discusses how to query hosts from the command line.

Use the `opcserverinfo` function to make a query from the command line.

```
hostInfo = opcserverinfo('localhost')
```

```
hostInfo = 
                 Host: 'localhost'
             ServerID: {1x3 cell}
    ServerDescription: {1x3 cell}
     OPCSpecification: {'DA2' 'DA2' 'DA2'}
    ObjectConstructor: {1x3 cell}
```

Examining the returned structure in more detail provides the server IDs of each OPC server.

```
allServers = hostInfo.ServerID'
```

```
allServers = 
    'Matrikon.OPC.Simulation.1'
    'ICONICS.Simulator.1'
    'Softing.OPCToolboxDemo_ServerDA.1'
```

Step 2: Create an OPC Data Access Client Object

After determining the host name and server ID of the OPC server to connect to, you can create an `opcda` client object. The client controls the connection status to the server, and stores any events that occur from that server (such as notification of data changing state, which is called a *data change event*) in the event log. The `opcda` client object also contains any Data Access Group objects that you create on the client. For details on the OPC object hierarchy, see “Toolbox Object Hierarchy for the Data Access Standard” on page 6-2.

Use the `opcda` function to specify the host name and Server ID.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1')
```

```
da = 
   OPC Data Access Object: localhost/Matrikon.OPC.Simulation.1
      Server Parameters
         Host:           localhost
         ServerID:       Matrikon.OPC.Simulation.1
         Status:         disconnected
      Object Parameters
         Group:          0-by-1 dagroup object
```

For details on creating clients, see “Create OPC Data Access Objects” on page 6-2.

Step 3: Connect to the OPC Data Access Server

OPC Data Access Client objects are not automatically connected to the server when they are created. This allows you to fully configure an OPC object hierarchy (a client with groups and items) before connecting to the server, or without a server even being present.

Use the `connect` function to connect an `opcda` client object to the server at the command line.

```
connect(da)
```

Step 4: Create an OPC Data Access Group Object

You create Data Access Group objects (`dagroup` objects) to control and contain a collection of Data Access Item objects (`daitem` objects). A `dagroup` object controls how often the server must notify you of any changes in the item values, controls the activation status of the items in that group, and defines, starts, and stops logging tasks.

On their own, `dagroup` objects are not useful. Once you add items to a group, you can control those items, read values from the server for all the items in a group, and log data for those items, using the `dagroup` object. In Step 5 you browse the OPC server for available tags. Step 6 involves adding the items associated with those tags to the `dagroup` object.

Use the `addgroup` function to create `dagroup` objects from the command line. This example adds a group to the `opcda` client object already created.

```
grp = addgroup(da)
```

```
grp =
   OPC Group Object: Group0
      Object Parameters
         GroupType:        private
         Item:             0-by-1 daitem object
         Parent:           localhost/Matrikon.OPC.Simulation.1
         UpdateRate:       0.5
         DeadbandPercent:  0
      Object Status
         Active:           on
         Subscription:     on
         Logging:          off
         LoggingMode:      memory
```

See “Create Data Access Group Objects” on page 6-4 for more information on creating group objects from the command line.

Step 5: Browse the Server Name Space

All OPC servers provide access to server items via a server name space. The name space is an ordered list of the server items, usually arranged in a hierarchical format for easy access. A server item (also known as a *tag*) is a measurement or data point on a server, providing information from a device (such as a pressure sensor) or from another software package that supplies data through OPC Data Access (such as a SCADA package).

Note If you know the item IDs of the server items you are interested in, you can skip this section and go directly to “Step 6: Add OPC Data Access Items to the Group” on page 2-5. In this example, assume that you do not know the exact item IDs, although you do know that you want to log information from the Saw-toothed Waves and Triangular Waves provided by the Matrikon Simulation Server.

From the command line, you can “browse” the server name space using the `serveritems` function. You need to supply a connected `opcda` client object to the `serveritems` function, and an optional character vector argument to limit the returned results. The character vector can contain wildcard characters (*). An example of using `serveritems` is as follows.

```
sawtoothItems = serveritems(da,'*Saw*')
```

```
sawtoothItems =
    'Saw-toothed Waves.'
    'Saw-toothed Waves.Int1'
    'Saw-toothed Waves.Int2'
    'Saw-toothed Waves.Int4'
    'Saw-toothed Waves.Money'
    'Saw-toothed Waves.Real4'
    'Saw-toothed Waves.Real8'
    'Saw-toothed Waves.UInt1'
    'Saw-toothed Waves.UInt2'
    'Saw-toothed Waves.UInt4'
```

The command for obtaining the server item properties is `serveritemprops`. See the `serveritemprops` reference page for details.

Step 6: Add OPC Data Access Items to the Group

Now that you have found the server items in the name space, you can add Data Access Item objects (`daitem` object) for those tags to the `dagroup` object you created in Step 4. A `daitem` object is a link to a tag in the name space, providing the tag value, and additional information on that item, such as the Canonical Data Type.

Reading a Value from the Server

A `daitem` object initially contains no information about the server item that it represents. The `daitem` object only updates when the server notifies the client of a change in status for that item (the notification is called a data change event) or the client specifically reads a value from the server.

Each time you read or obtain data from the server through a data change event, the server provides you with updated Value, Quality, and Timestamp values.

Adding More Items to the Group

Use the `additem` function to add items to a `dagroup` object. You need to pass the `dagroup` object to which the items will be added, and the fully qualified item ID as a character vector. The item IDs were found using the `serveritems` function in Step 5.

```
itm1 = additem(grp,'Saw-toothed Waves.Real8')
```

```
itm1 =
   OPC Item Object: Saw-toothed Waves.Real8
      Object Parameters
         Parent:           Group0
         AccessRights:     read/write
         DataType:         double
      Object Status
         Active:           on
      Data:
         Value:
         Quality:
         Timestamp:
```

You can add multiple items to the group in one `additem` call, by specifying multiple `ItemID` values in a cell array.

```
itms = additem(grp,{'Triangle Waves.Real8', ...
                    'Triangle Waves.UInt2'})
```

```
itms =
   OPC Item Object Array:
   Index:  DataType:  Active:  ItemID:
   1       double     on       Triangle Waves.Real8
   2       uint16     on       Triangle Waves.UInt2
```

For details on adding items to groups, see “Create Data Access Item Objects” on page 6-6.

Step 7: View All Item Values

The group object lets you read and write values from all items in the group, and log data to memory and/or disk.

The **Value**, **Quality**, and **Timestamp** values of items continually update as long as you have **Subscription** enabled. Subscription controls whether *data change events* are sent by the OPC server to the toolbox, for items whose values change. UpdateRate and DeadbandPercent define how often the items must be queried for a new value, and whether all value changes or only changes of a specified magnitude are sent to the toolbox. For details on Subscription, see “Data Change Events and Subscription” on page 7-8.

By observing the data for a while, you will see that the three signals appear to have similar ranges. This indicates that you can visualize the data in the same axes when you plot it in Step 10.

In Step 9 you will configure a logging task and log data for the three items.

Use the `read` function with a group object as the first parameter to read values from all items in a group. The `read` function is discussed in detail in “Read and Write Data on OPC DA Server” on page 7-2.

Step 8: Configure Group Properties for Logging

Now that your `dagroup` object contains items, use the group to control the interaction of those items with the server. In this step, configure the group to log data from those items for 2 minutes at 0.2-second intervals. You can use the logged data in Step 9 to visualize the signals produced by the Matrikon Simulation Server.

OPC Data Access Servers provide access only to “live” data (the last known value of each server item in their name space). In many cases, a single value of a signal is not useful, and a time series containing the signal value over a period of time is helpful in analyzing that signal or signal set. Industrial Communication Toolbox allows you to log all OPC DA items in a group to disk or memory, and to retrieve that data for analysis in MATLAB.

You configure a logging session using the `dagroup` object. By modifying the properties associated with logging, you control how often the data must be sent from the server to the client, how many records the group must log, and where to log the data.

Use the `set` function to set OPC object properties. From the command line you can calculate the number of records required for the logging task.

```
logDuration = 2*60;
logRate = 0.2;
numRecords = ceil(logDuration./logRate);
grp.UpdateRate = logRate;
grp.RecordsToAcquire = numRecords;
```

Step 9: Log OPC Server Data

Now that you configured the `dagroup` object's logging properties, your object can log the required amount of data to memory.

Use the `start` function with the required `dagroup` object to start a logging task.

```
start(grp)
```

The logging task occurs in the background. You can continue working in MATLAB while a logging task is in operation. The logging task is unaffected by other computations occurring in MATLAB, and MATLAB processing is not blocked by the logging task. You can instruct MATLAB to wait for the logging task to complete, using the `wait` function.

```
wait(grp)
```

Step 10: Plot the Data

After logging finishes, transfer data from the toolbox engine to the MATLAB workspace using the `getdata` function, which provides two types of output, depending on its `'datatype'` argument. For details, see the `getdata` reference page. In this case you retrieve the data into separate arrays, and plot the data.

This example produces the figure:

```
[logIDs, logVal, logQual, logTime, logEvtTime] = ...
    getdata(grp,'double');
plot(logTime,logVal)
axis tight
datetick('x','keeplimits')
legend(logIDs)
```

Notice how the three signals seem almost completely unrelated, except for the period of the two `Real8` signals. The peak values for each signal are different, as are the periods for the two `Triangle Waves` signals. By visualizing the data, you can gain some insight into the way the Matrikon OPC Simulation Server simulates each tag. In this case, it is apparent that `Real8` and `UInt2` signals have a different period.

Step 11: Clean Up

After finishing an OPC task, you should remove the task objects from memory and clear the MATLAB workspace of the variables associated with these objects.

When using OPC objects at the MATLAB command line or from your own functions, you must remove them from the toolbox engine using the `delete` function. Note that when you delete a toolbox object, the children of that object are automatically removed from the toolbox engine. In this example, there is no need to delete `grp` and `itm`, as they are children of `da`.

```
disconnect(da)
delete(da)
clear da grp itm
close(gcf)
```

OPC object management is discussed in detail in “Delete Objects” on page 6-18.

Quick Start: Using the OPC Data Access Explorer

The best way to learn about the OPC capabilities in Industrial Communication Toolbox is to look at a simple example. This topic shows the basic steps required to log data from an OPC data access server for analysis and visualization. The example uses the **OPC Data Access Explorer** app provided in the toolbox, to show the process, and includes information on how to achieve the same results from the command line.

This topic contains cross-references to other sections in the documentation that provide more in-depth discussions of the relevant concepts.

Access Data with the OPC Data Access Explorer

In this section...

"Procedure Overview" on page 3-2
"Step 1: Open the OPC Data Access Explorer" on page 3-2
"Step 2: Locate Your OPC Server" on page 3-3
"Step 3: Create an OPC Data Access Client Object" on page 3-5
"Step 4: Connect to the OPC Server" on page 3-7
"Step 5: Create an OPC Data Access Group Object" on page 3-8
"Step 6: Browse the Server Name Space" on page 3-10
"Step 7: Add OPC Data Access Items to the Group" on page 3-12
"Step 8: View All Item Values" on page 3-14
"Step 9: Configure Group Properties for Logging" on page 3-15
"Step 10: Log OPC Server Data" on page 3-17
"Step 11: Plot the Data" on page 3-17
"Step 12: Clean Up" on page 3-19

Procedure Overview

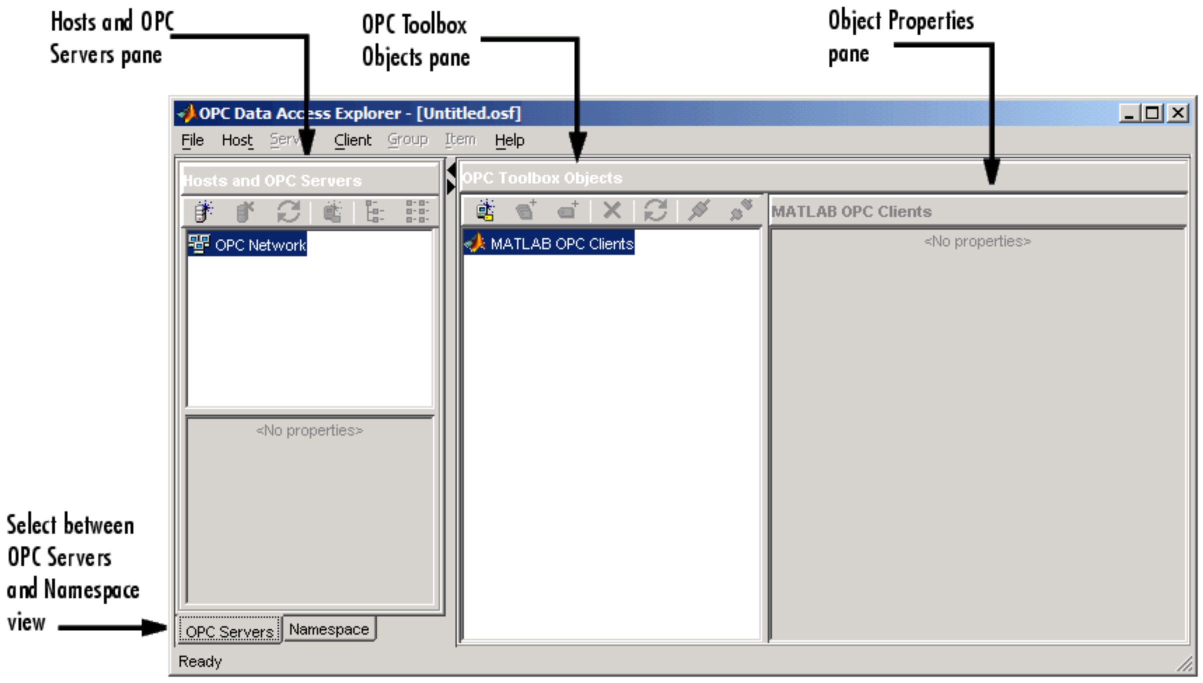
This section illustrates the basic steps required to create an OPC Data Access application by visualizing the Triangle Wave and Saw-toothed Wave signals provided with the Matrikon OPC Simulation Server. The application logs data to memory and plots that data, highlighting uncertain or bad data points. By visualizing the data you can more clearly see the relationships between the signals.

Note To run the sample code in the following examples, you must have the Matrikon OPC Simulation Server available on your local machine. For information on installing this, see "Install an OPC DA or HDA Simulation Server for OPC Classic Examples" on page 1-14. The code requires only minor changes to work with other servers.

The example in this topic uses the **OPC Data Access Explorer** app. In addition, each step contains information on how to complete that step using command-line code. The entire example is contained in the example file `opcdemo_quickstart`.

Step 1: Open the OPC Data Access Explorer

Double-click the **OPC Data Access Explorer** in the Apps menu. The app opens with no hosts, servers, or toolbox objects created. The following figure shows the main components of the **OPC Data Access Explorer**.



In the following steps, you will fill each of the panes with information required to log data, and you will log the data, by creating and interacting with OPC objects.

Command-Line Equivalent

To open the **OPC Data Access Explorer** from the command line, type `opcDataAccessExplorer` at the MATLAB prompt.

Step 2: Locate Your OPC Server

In this step, you obtain two pieces of information that the toolbox needs to uniquely identify the OPC server that you want to access. You use this information when you create an OPC Data Access Client object (`opcda` client object), described in “Step 3: Create an OPC Data Access Client Object” on page 3-5.

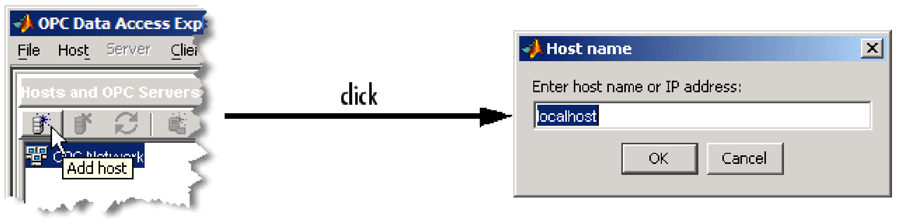
The first piece of information that you require is the hostname of the server computer. The hostname (a descriptive name like `PlantServer` or an IP address such as `192.168.16.32`) qualifies that computer on the network, and is used by the OPC Data Access protocols to determine the available OPC servers on that computer, and to communicate with the computer to establish a connection to the server. In any OPC application, you must know the name of the OPC server’s host, so that a connection with that host can be established. Your network administrator will be able to provide you with a list of hostnames that provide OPC servers on your network. In this example, you will use `localhost` as the hostname, because you will connect to the OPC server on the same machine as the client.

The second piece of information that you require is the OPC server’s *server ID*. Each OPC server on a particular host is identified by a unique server ID (also called the Program ID or ProgID), which is allocated to that server on installation. The server ID is a character vector, usually containing periods.

Although your network administrator will be able to provide you with a list of server IDs for a particular host, you can query the host for all available OPC servers. “Discover Available Data Access Servers” on page 5-2 discusses how to query hosts from the command line.

Using the **OPC Data Access Explorer** you can browse a host using the following steps:

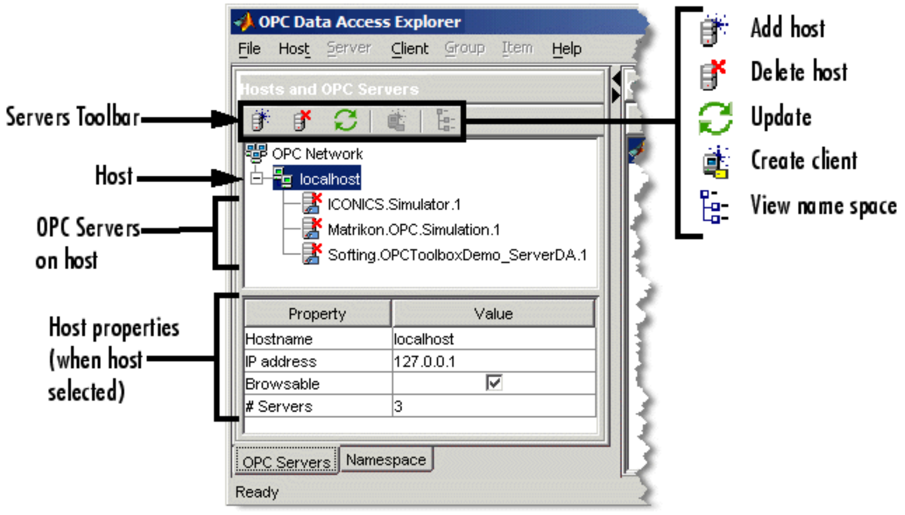
1 In the **Hosts and OPC Servers** pane, click the **Add host** icon to open the Host name dialog, shown below.



2 In the Host name dialog, enter the name of the host. In this case, you can use the "`localhost`" alias.

```
localhost
```

Click **OK**. The hostname will be added to the **OPC Network** tree view, and the OPC servers installed on that host will automatically be found and added to the tree view. Your **Hosts and OPC Servers** pane should look similar to the one shown below.



Property	Value
Hostname	localhost
IP address	127.0.0.1
Browsable	✓
# Servers	3

Note that the local host in this example provides three OPC servers. The Server ID for this example is `'Matrikon.OPC.Simulation.1'`.

Command-Line Equivalent

The command-line equivalent for this step uses the function `opcserverinfo`.

```
hostInfo = opcserverinfo('localhost')
```

```
hostInfo =
              Host: 'localhost'
          ServerID: {1x3 cell}
```

```
    ServerDescription: {1x3 cell}
     OPCSpecification: {'DA2'  'DA2'  'DA2'}
    ObjectConstructor: {1x3 cell}
```

Examining the returned structure in more detail provides the server IDs of each OPC server.

```
allServers = hostInfo.ServerID'
```

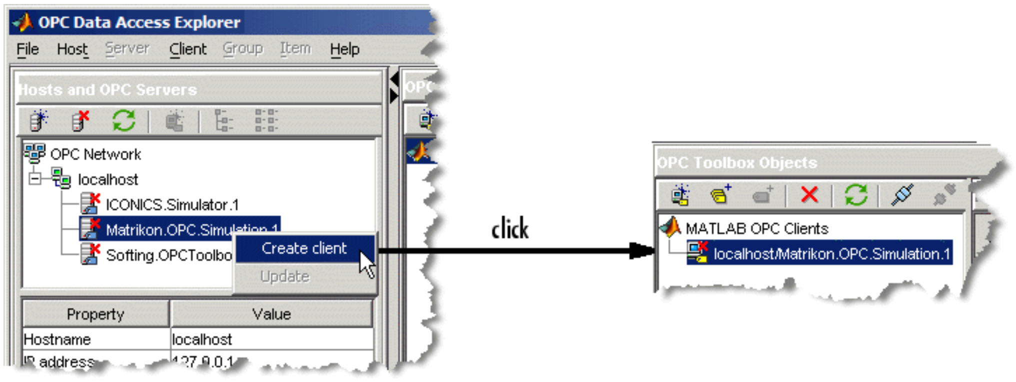
```
allServers =
    'Matrikon.OPC.Simulation.1'
    'ICONICS.Simulator.1'
    'Softing.OPCToolboxDemo_ServerDA.1'
```

Step 3: Create an OPC Data Access Client Object

Once you have determined the hostname and server ID of the OPC server you want to connect to, you can create an `opcda` client object. The client controls the connection status to the server, and stores any events that take place from that server (such as notification of data changing state, which is called a *data change event*) in the event log. The `opcda` client object also contains any Data Access Group objects that you create on the client. For more information on the OPC object hierarchy, see “Toolbox Object Hierarchy for the Data Access Standard” on page 6-2.

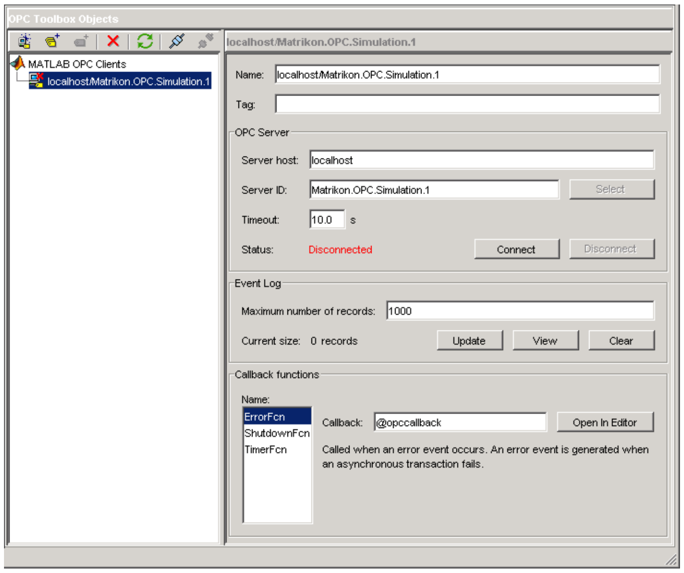
With the **OPC Data Access Explorer**, you can create a client directly from the **Hosts and OPC Servers** pane.

Right-click the `Matrikon` server node and choose **Create client**. A client will be created in the **OPC Toolbox Objects** pane, as shown in the following figure.



The name of the client (displayed in the **OPC Toolbox Objects** pane) is `*Host/ServerID*`, where `*Host*` is the hostname and `*ServerID*` is the Server ID associated with that client. In this example, the client’s name is `'localhost/Matrikon.OPC.Simulation.1'`

Once you have created the client, you can view the properties of the client object in the **Object Properties** pane, as shown in the next figure.



OPC Toolbox Objects

MATLAB OPC Clients
localhost/Matrikon.OPC.Simulation.1

localhost/Matrikon.OPC.Simulation.1

Name: localhost/Matrikon.OPC.Simulation.1

Tag:

OPC Server

Server host: localhost

Server ID: Matrikon.OPC.Simulation.1 Select

Timeout: 10.0 s

Status: Disconnected Connect Disconnect

Event Log

Maximum number of records: 1000

Current size: 0 records Update View Clear

Callback functions

Name:
ErrorFcn
ShutdownFcn
TimerFcn

Callback: @opccallback Open In Editor

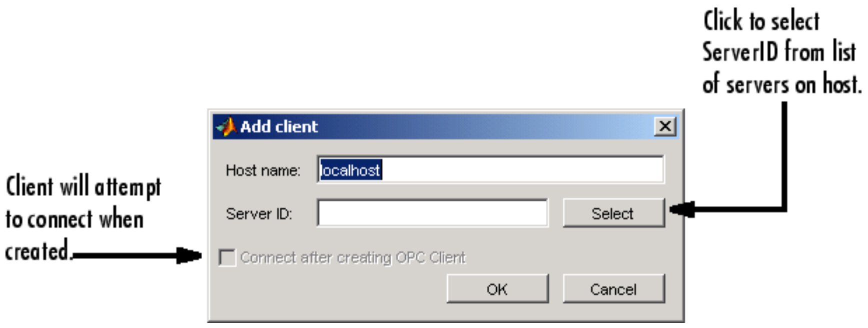
Called when an error event occurs. An error event is generated when an asynchronous transaction fails.

Alternative Methods for Creating Clients

You can create a client in the **OPC Data Access Explorer** by using any of the following methods:

- Select the **MATLAB OPC Clients** node in the **OPC Toolbox Objects** pane and click **Add Client** in the **OPC Toolbox Objects** toolbar.
- Choose **Add** from the **Client** menu.
- Right-click the **MATLAB OPC Clients** node in the **OPC Toolbox Objects** tree and select **Create Client**.

If you select one of these methods, a dialog appears requesting the hostname and server ID.


Click to select ServerID from list of servers on host.
Client will attempt to connect when created.

Add client

Host name: localhost

Server ID:

Select

Connect after creating OPC Client

OK Cancel

When you supply a hostname, you will be able to select the Server ID from a list, by clicking **Select**. Using the Add client dialog, you can also automatically attempt to connect to the server when the client is created, by checking **Connect after creating OPC Client** before clicking **OK**.

Command-Line Equivalent

The command-line equivalent of this step involves using the `opcda` function, specifying the hostname and Server ID arguments.

```
da = opcda('localhost', 'Matrikon.OPC.Simulation.1')
da =
   OPC Data Access Object: localhost/Matrikon.OPC.Simulation.1
      Server Parameters
         Host:          localhost
         ServerID:      Matrikon.OPC.Simulation.1
         Status:        disconnected
      Object Parameters
         Group:         0-by-1 dagroup object
```

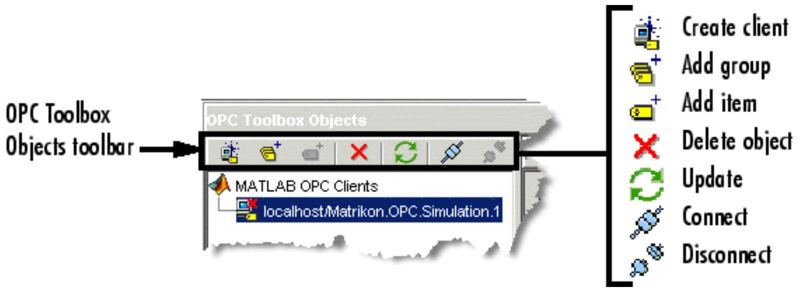
For more information on creating clients, see “Create OPC Data Access Objects” on page 6-2.

Step 4: Connect to the OPC Server

OPC Data Access Client objects are not automatically connected to the server when they are created. This allows you to fully configure an OPC object hierarchy (a client with groups and items) prior to connecting to the server, or without a server even being present.

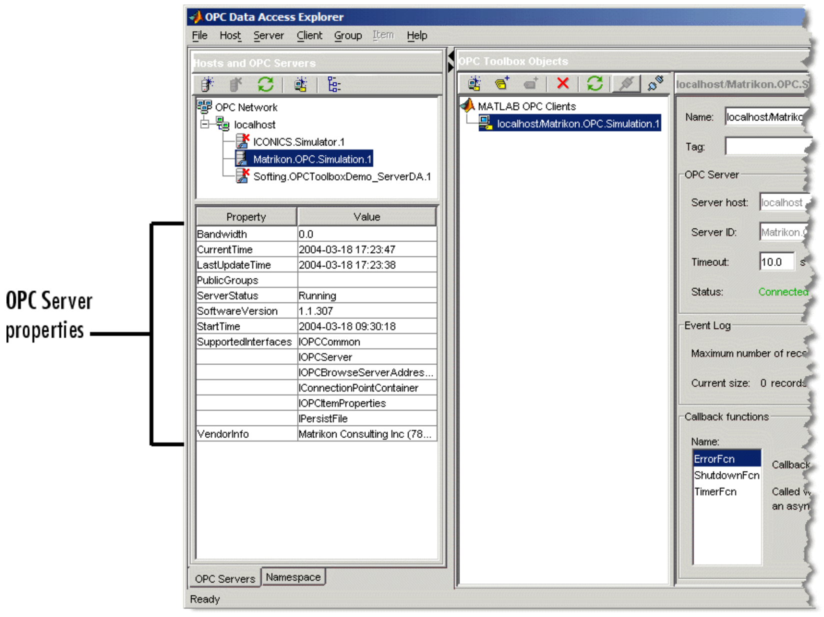
Note The Add Client dialog described in “Alternative Methods for Creating Clients” on page 3-6 can connect the client to the server after creating the client object.

To connect the client to the server, you can use the **OPC Toolbox Objects** toolbar, shown in the following figure.



Click **Connect** in the **OPC Toolbox Objects** toolbar. If the client is able to connect to the server, the icon for that client in the **OPC Toolbox Objects** tree will change to show that the client is connected. If the client could not connect to the server, an error dialog will show any error message returned. See “Troubleshooting OPC Issues” on page 1-17 for information on why a client may not be able to connect to a server.

When you connect an `opcda` client object to the server associated with that client, the server node in the **Hosts and OPC Servers** pane also updates to show that the server has a connection to a client in the app. With that connection, the properties of the server are displayed in the **Hosts and OPC Servers** pane. For this example, a typical view of the app after connecting to a client is shown in the next figure.



Property	Value
Bandwidth	0.0
CurrentTime	2004-03-18 17:23:47
LastUpdateTime	2004-03-18 17:23:38
PublicGroups	
ServerStatus	Running
SoftwareVersion	1.1.307
StartTime	2004-03-18 09:30:18
SupportedInterfaces	IOPCCommon
	IOPCServer
	IOPCBrowseServerAddres...
	IConnectionPointContainer
	IOPCItemProperties
	IPersistFile
VendorInfo	Matrikon Consulting Inc (78...

Name: localhost/Matriko
Tag:
Server host: localhost
Server ID: Matrikon.
Timeout: 10.0 s
Status: Connected
Maximum number of rece
Current size: 0 records
Name:
Callback
Called w
an asyn

The OPC server properties include diagnostic information, such as the supported OPC Data Access interfaces, the time the server was started, and the current server status.

Command-Line Equivalent

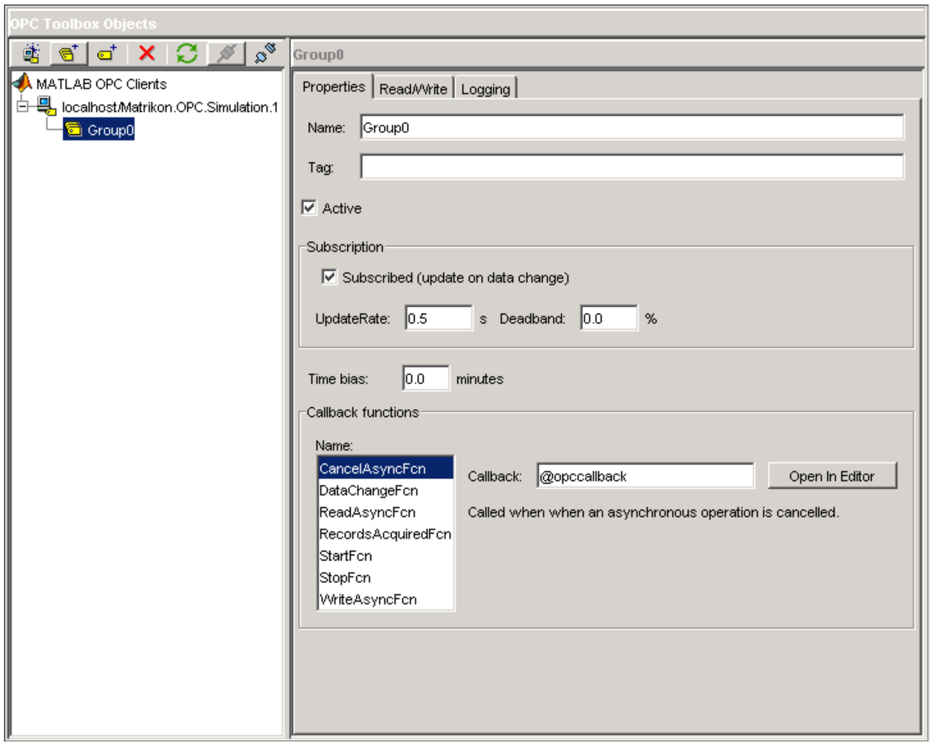
You use the `connect` function to connect an `opcda` client object to the server at the command line.

```
connect(da)
```

Step 5: Create an OPC Data Access Group Object

You create Data Access Group objects (`dagroup` objects) to control and contain a collection of Data Access Item objects (`daitem` objects). A `dagroup` object controls how often the server must notify you of any changes in the item values, control the activation status of the items in that group, and define, start, and stop logging tasks.

To create a `dagroup` object, click **Add group** in the **OPC Toolbox Objects** toolbar. A group is created and automatically named, either by the OPC server or by the toolbox.



Name: Group0
Tag:
☑ Active
Subscription
☑ Subscribed (update on data change)
UpdateRate: 0.5 s Deadband: 0.0 %
Time bias: 0.0 minutes
Callback functions
Name: CancelAsyncFcn, DataChangeFcn, ReadAsyncFcn, RecordsAcquiredFcn, StartFcn, StopFcn, WriteAsyncFcn
Callback: @opccallback
Called when when an asynchronous operation is cancelled.

On their own, `dagroup` objects are not useful. Once you add items to a group, you can control those items, read values from the server for all the items in a group, and log data for those items, using the `dagroup` object. In Step 6 you browse the OPC server for available tags. Step 7 involves adding the items associated with those tags to the `dagroup` object.

Command-Line Equivalent

To create `dagroup` objects from the command line, you use the `addgroup` function. This example adds a group to the `opcda` client object already created.

```
grp = addgroup(da)
```

```
grp =
   OPC Group Object: Group0
      Object Parameters
         GroupType:        private
         Item:             0-by-1 daitem object
         Parent:           localhost/Matrikon.OPC.Simulation.1
         UpdateRate:       0.5
         DeadbandPercent:  0
      Object Status
         Active:           on
         Subscription:     on
         Logging:          off
         LoggingMode:      memory
```

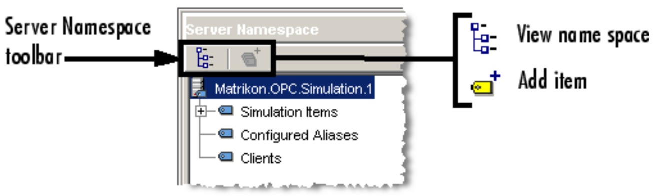
See “Create Data Access Group Objects” on page 6-4 for more information on creating group objects from the command line.

Step 6: Browse the Server Name Space

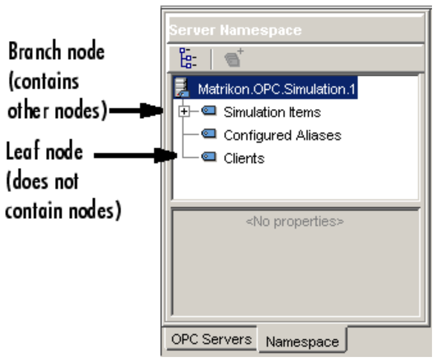
All OPC servers provide access to server items via a server name space. The name space is an ordered list of the server items, usually arranged in a hierarchical format for easy access. A server item (also known as a *tag*) is a measurement or data point on a server, providing information from a device (such as a pressure sensor) or from another software package that supplies data through OPC Data Access (such as a SCADA package).

Note If you know the item IDs of the server items you are interested in, you can skip this section and proceed to “Step 7: Add OPC Data Access Items to the Group” on page 3-12. In this example, assume that you do not know the exact item IDs, although you do know that you want to log information from the Saw-toothed Waves and Triangular Waves provided by the Matrikon Simulation Server.

The **Namespace** tab of the **Hosts and Servers** pane allows you to graphically browse a server’s name space. Because most OPC servers contain thousands of server items, retrieving a name space can be time consuming. When you connect to a server for the first time, the name space is not automatically retrieved. You have to request the name space using one of the **View** buttons in the **Server Namespace** toolbar, as shown in the following figure.



Click *View hierarchical namespace* to retrieve the hierarchical name space for the Matrikon OPC Server. A tree view containing the Matrikon name space is shown in the pane. Your pane should look similar to the following figure.


Branch node (contains other nodes)
Leaf node (does not contain nodes)
Server Namespace
Matrikon.OPC.Simulation.1
Simulation Items
Configured Aliases
Clients
<No properties>
OPC Servers
Namespace

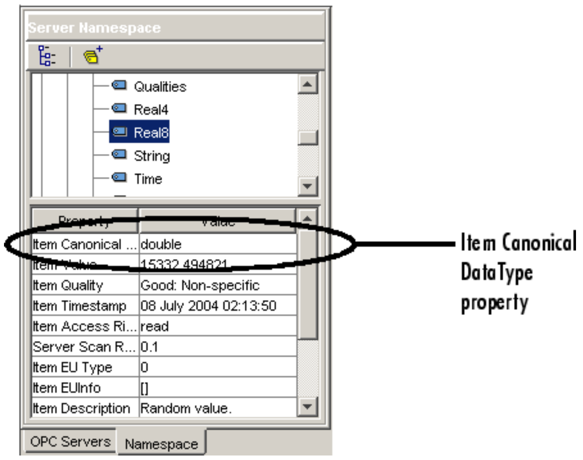
Note If you choose to view the name space as flat, you get a single list of all server items in the name space, expanded to their fully qualified names. A fully qualified name can be used to create a `daitem` object.

Browsing the name space using the app also provides some property information for each server item. The properties include the published OPC Item properties such as Value, Quality, and

Timestamp, plus additional properties published by the OPC server that may provide more information on that particular server item. For a list of standard OPC properties and an explanation of their use, see “OPC DA Server Item Properties” on page B-2.

In this example, you need to locate the Saw-toothed Waves and Triangle Waves signals in the Matrikon Simulation Server. You can achieve this using the following steps:

1. Ensure that you are viewing the hierarchical name space.
2. Expand the **Simulation items** node. You will see all the signal types that the Matrikon Server simulates.
3. Expand the **Saw-toothed Waves** node. A number of *leaf* nodes appear. A leaf node contains no other nodes, and usually signifies a tag on an OPC server.
4. Select the `Real8` leaf node. The properties of the server item appear in the properties table below the name space tree, as shown in the following figure.



Property	Value
Item Canonical ...	double
Item Value	15332.494821
Item Quality	Good: Non-specific
Item Timestamp	08 July 2004 02:13:50
Item Access Ri...	read
Server Scan R...	0.1
Item EU Type	0
Item EUInfo	[]
Item Description	Random value.

   Note the `Item Canonical DataType` property, which is `double`. The Canonical DataType is the data type that the server uses to store the server item’s value.

5. Select the `UInt2` leaf node. You will notice that the properties update, and the `Item Canonical Datatype` property for this server item is `uint16`. (MATLAB denotes integers with the number of bits in the integer, such as `uint16`; the Matrikon Server uses the COM Variant convention denoting the number of bytes, such as `UInt2`.)

You can continue browsing the server name space using the **Server Namespace** pane in the app. One unique characteristic of the Matrikon Simulation Server is that you can view the connected clients through the name space, by selecting the **Clients** node in the root of the name space.

In Step 7, you add three items to your newly created group object, using the **Server Namespace** pane.

Command-Line Equivalent

From the command line, you can browse the server name space using the `serveritems` function. You need to supply a connected `opcda` client object to the `serveritems` function, and an optional character vector argument to limit the returned results. The character vector can contain wildcard characters (*). An example of using `serveritems` is as follows.

```
sawtoothItems = serveritems(da, '*Saw*')
sawtoothItems = 
    'Saw-toothed Waves.'
    'Saw-toothed Waves.Int1'
    'Saw-toothed Waves.Int2'
    'Saw-toothed Waves.Int4'
    'Saw-toothed Waves.Money'
    'Saw-toothed Waves.Real4'
    'Saw-toothed Waves.Real8'
    'Saw-toothed Waves.UInt1'
    'Saw-toothed Waves.UInt2'
    'Saw-toothed Waves.UInt4'
```

The command-line equivalent for obtaining the server item properties is `serveritemprops`. See the `serveritemprops` reference page for more information on using the function.

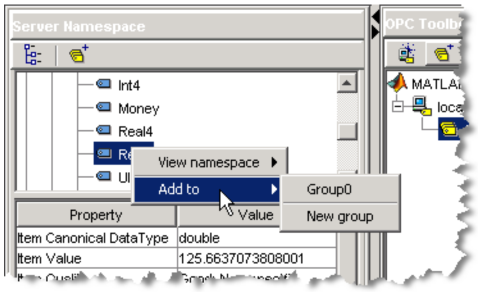
Step 7: Add OPC Data Access Items to the Group

Now that you have found the server items in the name space, you can add Data Access Item objects (`daitem` object) for those tags to the `dagroup` object you created in Step 5. A `daitem` object is a link to a tag in the name space, providing the tag value, and additional information on that item, such as the Canonical Data Type.

Using the app, you create items directly from the name space tree, using a context menu on each node in the tree.

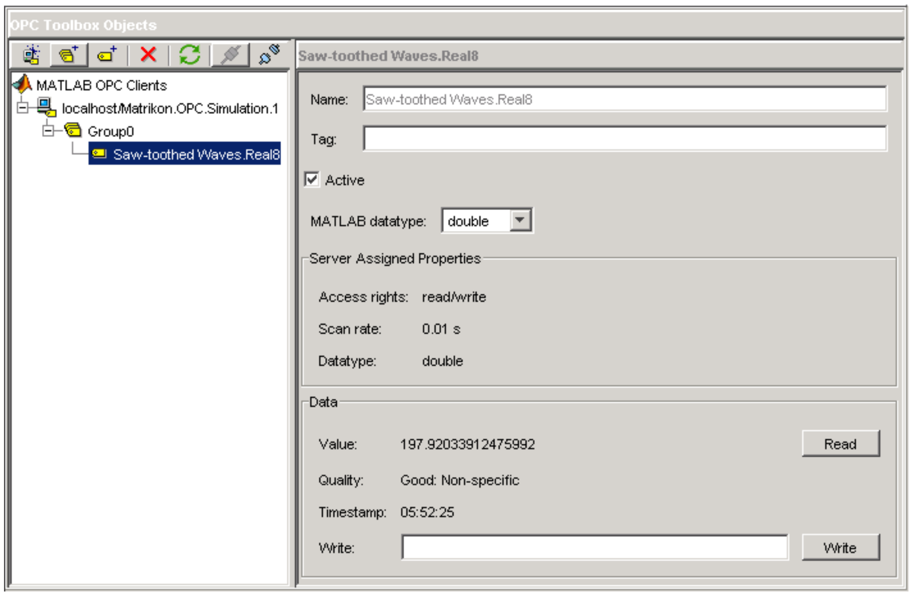
Browse to **Simulated Items** > **Saw-toothed Waves** > **Real8**, and right-click that node to bring up the context menu. Selecting **Add to** from the context menu provides you with a list of created groups for the item associated with that server, and a menu item to create a **New group** (and add the item to that group).

The menu displayed for this example is shown in the following figure.



Property	Value
Item Canonical DataType	double
Item Value	125.6637073808001

Click **Group0** to add the item to the already existing group that you created in Step 5. A `daitem` object is created in the **OPC Toolbox Objects** pane. The following figure shows the newly created item highlighted, with the properties of the item shown in the **Properties** pane.



Saw-toothed Waves.Real8

Name: Saw-toothed Waves.Real8

Tag:

☑ Active

MATLAB datatype: double

Server Assigned Properties

Access rights: read/write

Scan rate: 0.01 s

Datatype: double

Data

Value: 197.92033912475992

Quality: Good: Non-specific

Timestamp: 05:52:25

Write:

Read a Value from the Server

A `daitem` object initially contains no information about the server item that it represents. The `daitem` object only updates when the server notifies the client of a change in status for that item (the notification is called a data change event) or the client specifically reads a value from the server. Using the app, you can force a read of the item by clicking **Read** in the **Properties** pane of the required item.

Click **Read**. The **Value**, **Quality**, and **Timestamp** fields in the app will update. **Value** contains the last value that the server read from that particular item. **Quality** provides a measure of how meaningful **Value** is. If **Quality** is `Good`, then the **Value** can be trusted to be the same as the device or object to which the item refers, but only at the time provided by the **Timestamp** field. If **Quality** is anything other than `Good`, then the **Value** of the item is questionable.

Each time you read or obtain data from the server through a data change event, the server will provide you with updated Value, Quality, and Timestamp values.

Add More Items to the Group

Using the **Namespace** pane, expand the **Triangle Waves** node and add items for the **Real8** and **UInt2** server items. You will then have three items associated with your `dagroup` object. In Step 8, you configure a logging session for that group. You then log data in Step 9 from the three items you just created, and visualize the data in Step 10.

Command-Line Equivalent

You use the `additem` function to add items to a `dagroup` object. You need to pass the `dagroup` object to which the items will be added, and the fully qualified item ID as a character vector. The item IDs were found using the `serveritems` function in Step 6.

```
itm1 = additem(grp, 'Saw-toothed Waves.Real8')
```

```
itm1 =
   OPC Item Object: Saw-toothed Waves.Real8
      Object Parameters
```

```
         Parent:           Group0
         AccessRights:     read/write
         DataType:         double
      Object Status
         Active:           on
      Data:
         Value:
         Quality:
         Timestamp:
```

You can add multiple items to the group in one `additem` call, by specifying multiple `ItemID` values in a cell array.

```
itms = additem(grp, {'Triangle Waves.Real8', ...
'Triangle Waves.UInt2'})
```

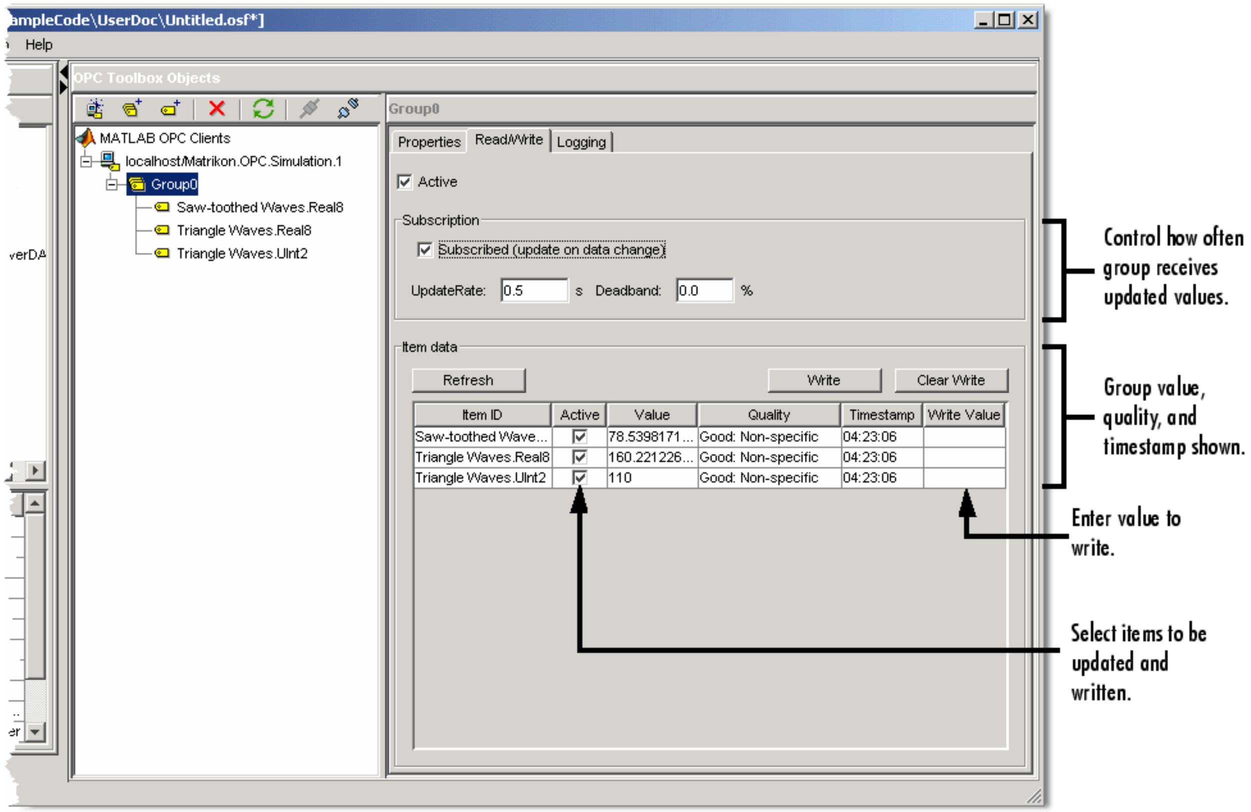
```
itms =
   OPC Item Object Array:
   Index:  DataType:  Active:  ItemID:
   1       double     on       Triangle Waves.Real8
   2       uint16     on       Triangle Waves.UInt2
```

For more information on adding items to groups, see “Create Data Access Item Objects” on page 6-6.

Step 8: View All Item Values

You can view the Value, Quality, and Timestamp for each item using the item properties pane. However, that view only provides access to one item at a time. The group object is designed to allow you to read and write values from all items in the group, and to log data to memory and/or disk. You use the **Group Read/Write** pane to view the values of the items you created in Step 7 to determine the approximate range of values that each item value varies over. The information from this pane will help you to verify that the data is updating, and whether you can plot the data in one set of axes or in subplots.

Click **Group0** in the **OPC Toolbox Objects** pane. Select the **Read/Write** tab in the top of the **Group properties** pane. The **OPC Toolbox Objects** pane should now look similar to the one shown in the following figure.



Item ID	Active	Value	Quality	Timestamp	Write Value
Saw-toothed Wave...	✓	78.5398171...	Good: Non-specific	04:23:06	
Triangle Waves.Real8	✓	160.221226...	Good: Non-specific	04:23:06	
Triangle Waves.UInt2	✓	110	Good: Non-specific	04:23:06	

The **Value**, **Quality**, and **Timestamp** values in the table of items will continually update as long as you have **Subscription** enabled. Subscription controls whether *data change events* are sent by the OPC server to the toolbox, for items whose values change. UpdateRate and DeadbandPercent define how often the items must be queried for a new value, and whether all value changes or only changes of a specified magnitude are sent to the toolbox. For more information on Subscription, see “Data Change Events and Subscription” on page 7-8.

By observing the data for a while, you will see that the three signals appear to have similar ranges. This indicates that you can visualize the data in the same axes when you plot it in Step 11.

You can also use the **Group Read/Write** pane for writing values to many items simultaneously. Specify a value in the **Write** column of the **Item data** table for each item you want to write to, and click **Write**, to be able to write to those items.

In Step 10 you will configure a logging task and log data for the three items.

Command-Line Equivalent

You can use the `read` function with a group object as the first parameter to read values from all items in a group. The `read` function is discussed in more detail in “Read and Write Data on OPC DA Server” on page 7-2.

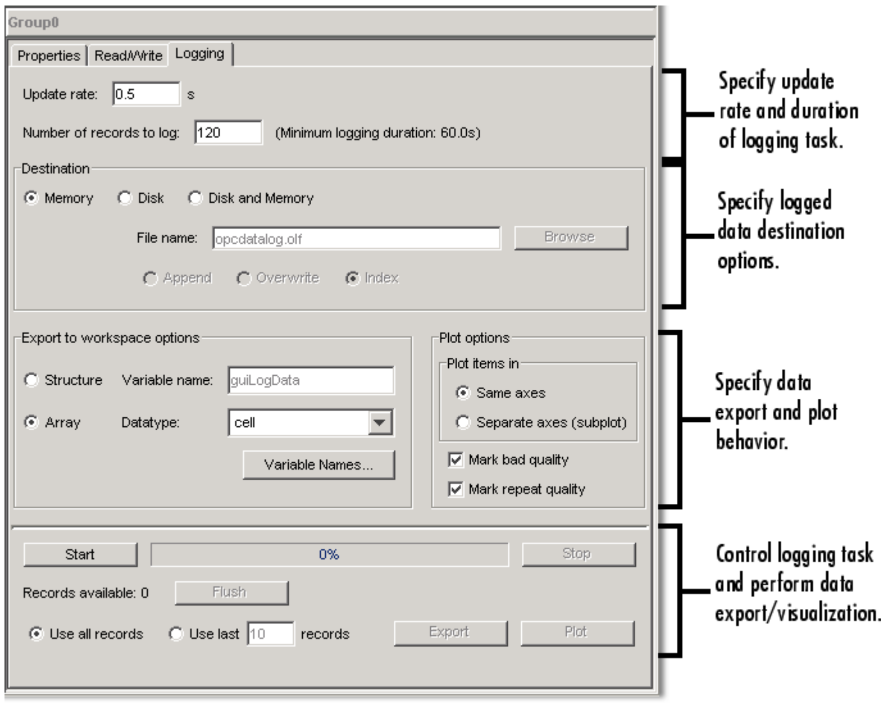
Step 9: Configure Group Properties for Logging

Now that your `dagroup` object contains items, you can use the group to control the interaction of those items with the server. In this step, you configure the group to log data from those items for 2 minutes at 0.2-second intervals. You will use the logged data in Step 11 to visualize the signals produced by the Matrikon Simulation Server.

OPC Data Access Servers provide access only to "live" data (the last known value of each server item in their name space). In many cases, a single value of a signal is not useful, and a time series containing the signal value over a period of time is helpful in analyzing that signal or signal set. Industrial Communication Toolbox allows you to log all OPC items in a group to disk or memory, and to retrieve that data for analysis in MATLAB.

You configure a logging session using the `dagroup` object. By modifying the properties associated with logging, you control how often the data must be sent from the server to the client, how many records the group must log, and where to log the data. This information is summarized in the **Logging** pane of the `dagroup` object properties in the app.

Select the **Logging** tab in the **Properties** pane. The following figure shows the **Logging** pane for the `dagroup` object created in this example.


Group0
Properties
Read/Write
Logging
Specify update rate and duration of logging task.
Specify logged data destination options.
Specify data export and plot behavior.
Control logging task and perform data export/visualization.
Browse
Variable Names...
Start
Stop
Flush
Export
Plot

Update rate: 0.5 s

Number of records to log: 120 (Minimum logging duration: 60.0s)

Destination

[✓ Memory] Disk Disk and Memory

File name: opcdatalog.olf

Append Overwrite [✓ Index]

Export to workspace options

Structure [✓ Array]

Variable name: guiLogData

Datatype: cell

Plot options

Plot items in

[✓ Same axes] Separate axes (subplot)

[✓ Mark bad quality]

[✓ Mark repeat quality]

0%

Records available: 0

[✓ Use all records] Use last 10 records

Using the **Logging** pane, configure the logging session using the following steps:

1 Set **Update rate** to `0.2`.
2 Set **Number of records to log** to `600`. Because you want to log for 2 minutes (120 seconds) at 0.2-second update rates, you need 600 (i.e., 120/0.2) records.

You can leave the rest of the logging properties at their default values, because this example uses data logged to memory.

In Step 10 you log the data. In Step 11 you will visualize the data.

Command-Line Equivalent

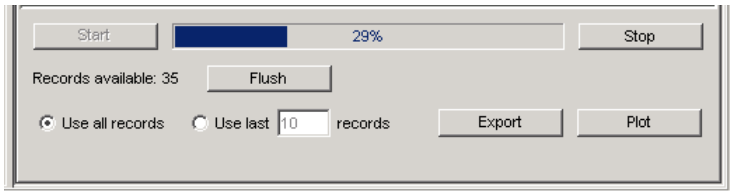
You use the `set` function to set OPC object properties. From the MATLAB command line, you can calculate the number of records required for the logging task.

```
logDuration = 2*60;
logRate = 0.2;
numRecords = ceil(logDuration./logRate)
set(grp, 'UpdateRate',logRate,'RecordsToAcquire',numRecords);
```

Step 10: Log OPC Server Data

In Step 9 you configured the `dagroup` object logging properties. Your object is now ready to log the required amount of data to memory.

Click **Start** in the **Logging** tab. The logging task begins, and the toolbox engine receives and stores the data from the OPC server. The progress bar indicates the status of the logging task, as shown in the following figure.



Note The logging task occurs in the background. You can continue working in MATLAB while a logging task is in operation. The logging task is not affected by any other computation taking place in MATLAB, and MATLAB is not blocked from processing by the logging task.

Wait for the task to complete before continuing with Step 11.

Command-Line Equivalent

You use the `start` function with the required `dagroup` object to start a logging task.

```
start(grp)
```

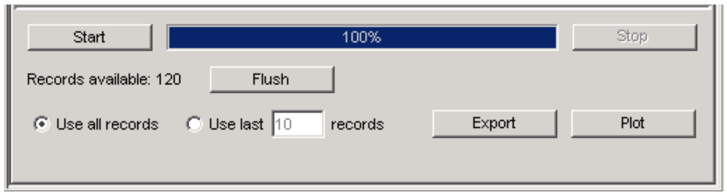
Although the logging operation takes place in the background, you can instruct MATLAB to wait for the logging task to complete, using the `wait` function.

```
wait(grp)
```

Step 11: Plot the Data

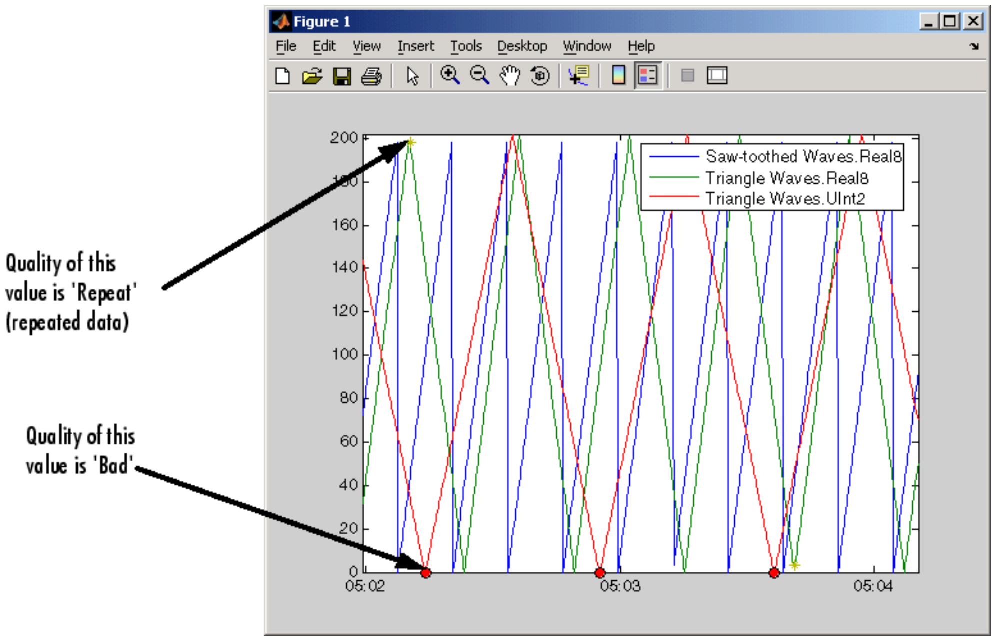
In this introductory example, you use the app to visualize the data logged in Step 10. In a more complex task, you would export the logged data to the workspace and use MATLAB functions to analyze and interpret the logged data.

When the logging task stops, the **Logging** pane will update to show that the task is complete. An example of the logging status portion of the **Logging** pane after a completed task is shown in the following figure.



To view the data from the app, click **Plot**. The logged data will be retrieved from the toolbox engine and displayed in a MATLAB figure window. The format of the displayed data and annotation options are controlled by settings in the **Plot** options frame of the **Logging** pane. By default, the plot will be annotated with any data points that have a Quality other than `Good`. Values whos Quality is `Bad` are annotated with a large red circle with a black border, and Values with Quality of `Repeat` are annotated with a yellow star. You should always view the Quality returned with the Value of an item to determine if the Value is meaningful or not. The relationship between the Value and Quality of an item is discussed in “OPC Data: Value, Quality, and TimeStamp” on page 8-2.

An example of the plotted data is shown in the next figure.



Note Your plotted data will almost certainly not look like the one shown here, because the logging task was executed at a different time.

Notice how the three signals seem almost completely unrelated, except for the period of the two **Real8** signals. The peak values for each signal are different, as are the periods for the two **Triangle Waves** signals. By visualizing the data, you can gain some insight into the way the Matrikon OPC Simulation Server simulates each tag. In this case, it is apparent that `Real8` and `UInt2` signals have a different period.

Command-Line Equivalent

When your logging task has completed you transfer data from the toolbox engine to the MATLAB workspace using the `getdata` function, which provides two types of output, depending on the `'datatype'` argument. For more information see `getdata` in the reference pages. In this case you retrieve the data into separate arrays, and plot the data.

The example below reproduces the figure display that you get when you click **Plot**.

```
[logIDs, logVal,logQual,logTime,logEvtTime] = ...
 getdata(grp,'double');
plot(logTime,logVal);
axis tight
datetick('x','keeplimits')
legend(logIDs)
```

Step 12: Clean Up

When you are finished with an OPC task, you should remove the task objects from memory and clear the MATLAB workspace of the variables associated with these objects. The **OPC Data Access Explorer** app automatically deletes all objects that it creates from the toolbox engine. If you work only in the **OPC Data Access Explorer**, you do not need to perform any further cleanup other than to close the app. You close the app by using the **Exit** option in the **File** menu, or by using the **Close** button in the title bar. You will be prompted to save the **OPC Data Access Explorer** session. You can choose to save the session to an OPC session file (`.osf` file) for later use, or exit without saving.

Command-Line Equivalent

When you use OPC objects from the MATLAB command line, or from your own functions, you must remove them from the OPC software engine using the `delete` function. Note that when you delete a toolbox object, the children of that object are automatically removed from the toolbox engine. In the following example, there is no need to delete `grp` and `itm`, as they are children of `da`.

```
disconnect(da)
delete(da)
clear da grp itm
close(gcf)
```

For more details about OPC object management, see “Delete Objects” on page 6-18.

Quick Start: Using OPC Historical Data Access Functions

The best way to learn about OPC capabilities is to look at a simple example. This chapter illustrates the basic steps required to read data from an OPC Data Historical Access (HDA) server for analysis and visualization.

This chapter references other sections in the documentation that provide detailed discussions of the relevant concepts.

Access Historical Data

In this section...

"HDA Programming Overview" on page 4-2

"Step 1: Locate Your OPC Historical Data Access Server" on page 4-2

"Step 2: Create an OPC Historical Data Access Client Object" on page 4-3

"Step 3: Connect to the OPC Historical Data Access Server" on page 4-3

"Step 4: Retrieve Historical Data" on page 4-3

"Step 5: Plot the Data" on page 4-4

"Step 6: Clean Up" on page 4-4

HDA Programming Overview

This section illustrates the basic steps to create an OPC Historical Data Access (HDA) application by retrieving historical data from the Triangle Wave and Saw-toothed Wave signals provided by the Matrikon OPC Simulation Server.

Note To run the sample code in the following steps you need the Matrikon OPC Simulation Server on your local machine. For installation details, see "Install an OPC DA or HDA Simulation Server for OPC Classic Examples" on page 1-14. The code requires only minor changes to work with other servers.

Step 1: Locate Your OPC Historical Data Access Server

In this step, you obtain two pieces of information that the toolbox needs to uniquely identify the OPC Historical Data Access server that you want to connect to. You use this information when creating an OPC Historical Data Access (HDA) client object, described in "Step 2: Create an OPC Historical Data Access Client Object" on page 4-3.

The first piece of information is the host name of the server computer. The host name (a descriptive name like "`HistorianServer`" or an IP address such as `192.168.16.32`) qualifies that computer on the network, and is used by the OPC protocols to determine the available OPC servers on that computer. In any OPC application, you must know the name of the OPC server's host, so that a connection with that host can be established. Your network administrator can provide a list of host names that provide OPC servers on your network. In this example, you will use `localhost` as the host name, because you will connect to the OPC server on the same machine as the client.

The second piece of information is the OPC server's *server ID*. Each OPC server on a particular host is identified by a unique server ID (also called the Program ID or ProgID), which is allocated to that server on installation. The server ID is a character vector, usually containing periods.

Although your network administrator can provide a list of server IDs for a particular host, you can query the host for all available OPC servers. "Discover Available HDA Servers" on page 12-4 discusses how to query hosts from the command line.

Use the `opchdaserverinfo` function to query from the command line.

```
hostInfo = opchdaserverinfo('localhost')

hostInfo = 
             1x4 OPC HDA ServerInfo array:
index    Host                ServerID          HDASpecification             Description
-----  ---------  ---------------------------------  -----  ------------------------------------------------
  1    localhost  Advosol.HDA.Test.3                 HDA1        Advosol HDA Test Server V3.0
  2    localhost  IntegrationObjects.OPCSimulator.1  HDA1        Integration Objects OPC DA DX HDA Simulator 2
  3    localhost  IntegrationObjects.OPCSimulator.1  HDA1        Integration Objects' OPC DA/HDA Server Simulator
  4    localhost  Matrikon.OPC.Simulation.1          HDA1        MatrikonOPC Server for Simulation and Testing
```

Examining the returned structure in more detail provides the server IDs of each OPC server.

```
allServers = {hostInfo.ServerID}

allServers = 
Columns 1 through 3
    'Advosol.HDA.Test.3'  'IntegrationObjects.OPCSimulator.1'  'IntegrationObjects.OPCSimulator.1'
Column 4
    'Matrikon.OPC.Simulation.1'
```

Step 2: Create an OPC Historical Data Access Client Object

After determining the host name and server ID of the OPC server to connect to, create an OPC HDA client object. The client controls the connection status to the server, and stores events that occur from that server.

Use the `opchda` function, specifying the host name and Server ID arguments.

```
hdaClient = opchda('localhost','Matrikon.OPC.Simulation.1')

hdaClient = 
OPC HDA Client localhost/Matrikon.OPC.Simulation.1:
          Host: localhost
      ServerID: Matrikon.OPC.Simulation.1
       Timeout: 10 seconds

        Status: disconnected

    Aggregates: -- (client is disconnected)
ItemAttributes: -- (client is disconnected)
Methods
```

For details on creating clients, see “Create an OPC HDA Client Object” on page 13-4.

Step 3: Connect to the OPC Historical Data Access Server

OPC Historical Data Access Client objects are not automatically connected to the server when they are created.

Use the `connect` function to connect an OPC HDA client object to the server at the command line.

```
connect(hdaClient)
```

Step 4: Retrieve Historical Data

Generate Historical Data

After connecting to the HDA server you can read historical data values for the `Saw-toothed Waves.Real8` and `Triangle Waves.Real8` items. The Matrikon Simulation Server stores data only for items that have been activated and read by an OPC Data Access client. For this reason, run this code to generate and automatically store data in the historian.

Enter the following at the command line:

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da);
additem(grp,'Saw-toothed Waves.Real8');
additem(grp,'Triangle Waves.Real8');
logDuration = 2*60;
logRate = 0.2;
numRecords = ceil(logDuration./logRate);
grp.UpdateRate = logRate;
grp.RecordsToAcquire = numRecords;
start(grp)
wait(grp)
```

Read a Value from the Historical Data Access Server

To read historical values from an HDA server for a particular time range, use the `readRaw` function. This function takes a list of items as well as a start and end time (demarcating the time span) for which historical data is required.

```
data = hdaClient.readRaw({'Saw-toothed Waves.Real8','Triangle Waves.Real8'},now-100000,now)

data =
1-by-2 OPC HDA Data object:
        ItemID                Value           Start TimeStamp           End TimeStamp             Quality
-----------------------  ----------------  -----------------------  -----------------------  ----------------------
Saw-toothed Waves.Real8  200 double values  2010-11-02 12:22:32.981  2010-11-02 12:23:13.363  1 unique quality [Raw]
Triangle Waves.Real8     199 double values  2010-11-02 12:22:33.141  2010-11-02 12:23:13.293  1 unique quality [Raw]
```

The retrieved historical data contains a Value, Timestamp, and Quality for each data point. To view these elements from the previous example, use the following instructions:

```
data.Value
data.TimeStamp
data.Quality
```

Step 5: Plot the Data

Use this code to generate the plot figure:

```
plot(data)
axis tight
datetick('x','keeplimits')
legend(data.ItemID)
```

Step 6: Clean Up

After using OPC objects at the MATLAB command line or from your own functions, you must remove them from the toolbox engine with the `delete` function. Note that when you delete a toolbox object, the children of that object are automatically removed from the toolbox engine.

```
disconnect(hda)
delete(hdaClient)
clear hdaClient data
```

Details of OPC object management are discussed in “Delete Objects” on page 6-18.

Data Access User's Guide

Introduction to OPC Data Access (DA)

- “Discover Available Data Access Servers” on page 5-2
- “Connect to OPC Data Access Servers” on page 5-4

Discover Available Data Access Servers

In this section...

"Prerequisites" on page 5-2
"Determine Server IDs for a Host" on page 5-2

Prerequisites

To interact with an OPC server, you need two pieces of information:

- The *hostname* of the computer on which the OPC server has been installed. Typically the hostname is a descriptive term (such as `'plantserver'`) or an IP address (such as `192.168.2.205`).
- The *server ID* of the server you want to access on that host. Because a single computer can host more than one OPC server, each OPC server installed on that computer is given a unique ID during the installation process.

Your network administrator will be able to provide you with the hostnames for all computers providing OPC servers on your network. You may also obtain a list of server IDs for each host on your network, or you can use the toolbox function `opcserverinfo` to access server IDs from a host, as described in the following section.

Determine Server IDs for a Host

When an OPC server is installed, a unique server ID must be assigned to that OPC server. The server ID provides a unique name for a particular instance of an OPC server on a host, even if multiple copies of the same server software are installed on the same machine.

To determine the server IDs of OPC servers installed on a host, call the `opcserverinfo` function, specifying the hostname as the only argument. When called with this syntax, `opcserverinfo` returns a structure containing information about all the OPC servers available on that host.

```
info = opcserverinfo('localhost')

info = 
                 Host: 'localhost'
             ServerID: {1x4 cell}
    ServerDescription: {1x4 cell}
     OPCSpecification: {'DA2'  'DA2'  'DA2'  'DA2'}
    ObjectConstructor: {1x4 cell}
```

The fields in the structure returned by `opcserverinfo` provide the following information.

Server Information Returned by opcserverinfo

Field	Description
`Host`	Character vector that identifies the name of the host. Note that no name resolution is performed on an IP address.
`ServerID`	Cell array containing the server IDs of all OPC servers accessible from that host.
`ServerDescription`	Cell array containing descriptive text for each server.
`OPCSpecification`	Cell array containing the OPC Specification that the server provides.
`ObjectConstructor`	Cell array containing default syntax you can use to create an OPC Data Access Client object associated with the server. See “Create a DA Client Object” on page 5-4 for more information.

Connect to OPC Data Access Servers

In this section...

“Overview” on page 5-4
“Create a DA Client Object” on page 5-4
“Connect a Client to the DA Server” on page 5-5
“Browse the OPC DA Server Name Space” on page 5-5

Overview

After you get information about your OPC servers, described in “Discover Available Data Access Servers” on page 5-2 you can establish a connection to the server by creating an OPC Client object and connecting that client to the server. These steps are described in the following sections.

Note To run the sample code in the following examples, you must have the Matrikon OPC Simulation Server available on your local machine. For information on installing this, see “Install an OPC DA or HDA Simulation Server for OPC Classic Examples” on page 1-14. The code requires only minor changes work with other servers.

Create a DA Client Object

To create an `opcda` object, call the `opcda` function specifying the hostname, and server ID. You retrieved this information using the `opcserverinfo` function (described in “Discover Available Data Access Servers” on page 5-2).

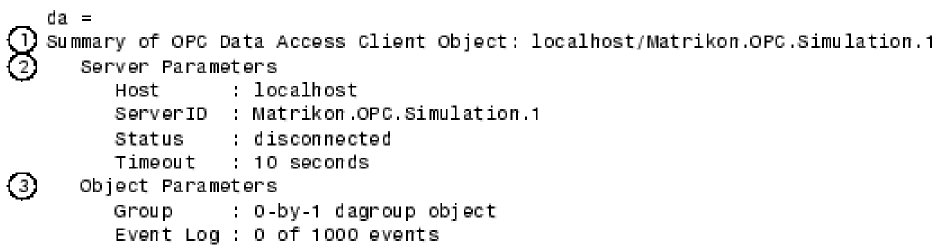
This example creates an `opcda` object to represent the connection to a Matrikon OPC Simulation Server. The `opcserverinfo` function includes the default `opcda` syntax in the `ObjectConstructor` field.

```
da = opcda('localhost', 'Matrikon.OPC.Simulation.1');
```

View a Summary of a Client Object

To view a summary of the characteristics of the `opcda` object you created, enter the variable name you assigned to the object at the command prompt. For example, this is the summary for the object `da`.

```
da
```



```
   da =
①  Summary of OPC Data Access Client Object: localhost/Matrikon.OPC.Simulation.1
②     Server Parameters
         Host      : localhost
         ServerID  : Matrikon.OPC.Simulation.1
         Status    : disconnected
         Timeout   : 10 seconds
③     Object Parameters
         Group     : 0-by-1 dagroup object
         Event Log : 0 of 1000 events
```

The items in this list correspond to the numbered elements in the object summary:

1. The title of the `Summary` includes the name of the `opcda` client object. The default name for a client object is made up of the `'host/serverID'`. You can change the name of a client object using the `set` function, described in “Configure OPC Data Access Object Properties” on page 6-13.
2. The `Server Parameters` provide information on the OPC server that the client is associated with. The host name, server ID, and connection status are provided in this section. You connect to an OPC server using the `connect` function, described in “Connect a Client to the DA Server” on page 5-5.
3. The `Object Parameters` section contains information on the OPC Data Access Group (`dagroup`) objects configured on this client. You use group objects to contain collections of items. Creating group objects is described in “Create Data Access Group Objects” on page 6-4.

Connect a Client to the DA Server

You connect a client to the server using the `connect` function.

```
connect(da);
```

Once you have connected to the server, the `Status` information in the client summary display will change from `'disconnected'` to `'connected'`.

If the client could not connect to the server for some reason (for example, if the OPC server is shut down) an error message will be generated. For information on troubleshooting connections to an OPC server, see “Troubleshooting OPC Issues” on page 1-17.

When you have connected the client to the server, you can perform the following tasks:

- Get diagnostic information about the OPC server, such as the server status, last update time, and supported interfaces. You use the `opcserverinfo` function to obtain this information.
- Browse the OPC server name space for information on the available server items. See “Browse the OPC DA Server Name Space” on page 5-5 for details.
- Create group and item objects to interact with OPC server data. See “Create OPC Data Access Objects” on page 6-2 for information.

Browse the OPC DA Server Name Space

A connected client object allows you to interact with the OPC server to obtain information about the name space of that server. The server name space provides access to all the data points provided by the OPC server by naming each of the data points with a server item, and then arranging those server items into a name space that provides a unique identifier for each server item.

This section describes how you use a connected client object to browse the name space and find information about each server item. These activities are described in the following sections:

- “Get the DA Server Name Space” on page 5-6 describes how to obtain a server name space, or a partial server name space, using the `getnamespace` and `serveritems` functions.
- “Get Information about a Specific Server Item” on page 5-7 describes how to query the server for the properties of a specific server item.

Get the DA Server Name Space

You use the `getnamespace` function to retrieve the name space from an OPC server. You must specify the client object that is connected to the server you are interested in. The name space is returned to you as a structure array containing information about each node in the name space.

The example below retrieves the name space of the Matrikon OPC Simulation Server installed on the local host.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
ns = getnamespace(da)

ns =
3x1 struct array with fields:
    Name
    FullyQualifiedID
    NodeType
    Nodes
```

The fields of the structure are described in the following table.

Field	Description
`Name`	The name of the node, as a character vector.
`FullyQualifiedID`	The fully qualified item ID of the node, as a character vector. The fully qualified item ID is made up of the path to the node, concatenated with `'.'` characters. You use the fully qualified item ID when creating an item object associated with this node.
`NodeType`	The type of node. `NodeType` can be `'branch'` (contains other nodes) or `'leaf'` (contains no other branches).
`Nodes`	Child nodes. `Nodes` is a structure array with the same fields as `ns`, representing the nodes contained in this branch of the name space.

From the example above, exploring the name space shows.

```
ns(1)

ans =
                Name: 'Simulation Items'
    FullyQualifiedID: 'Simulation Items'
            NodeType: 'branch'
               Nodes: [8x1 struct]

ns(3)

ans =
                Name: 'Clients'
    FullyQualifiedID: 'Clients'
            NodeType: 'leaf'
               Nodes: []
```

From the information above, the first node is a branch node called `'Simulation Items'`. Since it is a branch node, it is most likely not a valid server item. The third node is a leaf node (containing no other nodes) with a fully qualified ID of `'Clients'`. Since this node is a leaf node, it is most likely a server item that can be monitored by creating an item object.

To examine the nodes further down the tree, you need to reference the `Nodes` field of a branch node. For example, the first node contained within the `'Simulation Items'` node is obtained as follows.

```
ns(1).Nodes(1)

ans = 
               Name: 'Bucket Brigade'
    FullyQualifiedID: 'Bucket Brigade.'
           NodeType: 'branch'
              Nodes: [14x1 struct]
```

The returned result shows that the first node of `'Simulation Items'` is a branch node named `'Bucket Brigade'`, and contains 14 nodes.

```
ns(1).Nodes(1).Nodes(9)

ans = 
               Name: 'Real8'
    FullyQualifiedID: 'Bucket Brigade.Real8'
           NodeType: 'leaf'
              Nodes: []
```

The ninth node in `'Bucket Brigade'` is named `'Real8'` and has a fully qualified ID of `'Bucket Brigade.Real8'`. You use the fully qualified ID to refer to that specific node in the server name space when creating items.

You can use the `flatnamespace` function to flatten a hierarchical name space.

Get Information about a Specific Server Item

In addition to publishing a name space to all clients, an OPC server provides information about the properties of each of the server items in the name space. These properties provide information on the data format used by the server to store the server item value, a description of the server item, and additional properties configured when the server item was created. The additional properties can include information on the *range* of the server item, the maximum rate at which the server can update that server item value, etc. See “OPC DA Server Item Properties” on page B-2.

You access a property using a defined set of *property ID*s. A property ID is simply a number that defines a specific property of the server item. Property IDs are divided into three categories:

- “OPC Specific Properties” on page B-4 (1-99) that every OPC server must provide. The OPC Specific Properties include the server item’s Value, Quality, and Timestamp. For more information on understanding OPC data, see “OPC Data: Value, Quality, and TimeStamp” on page 8-2.
- “OPC Recommended Properties” on page B-5 (100-4999) that OPC servers can provide. These properties include maximum and minimum values, a description of the server item, and other commonly used properties..
- Vendor Specific Properties (5000 and higher) that an OPC server can define and use. These properties may be different for each OPC server, and provide a space for OPC server manufacturers to define their own properties.

You query properties of a server item using the `serveritemprops` function, specifying the client object, the fully qualified item ID of the server item you are interested in, and an optional vector of property IDs that you want to retrieve. If you do not specify the property IDs, all properties defined for that server item are returned.

Note You obtain the fully qualified item ID from the server using the `getnamespace` function or the `serveritems` function, which simply returns all fully qualified item IDs in a cell array of character vectors.

The following example queries the Item Description property (ID 101) of the server item `'Bucket Brigade.ArrayOfReal8'` from the example in “Get the DA Server Name Space” on page 5-6.

```
p = serveritemprops(da, 'Bucket Brigade.ArrayOfReal8', 101)

p = 
             PropID: 101
    PropDescription: 'Item Description'
          PropValue: 'Bucket brigade item.'
```

For a list of OPC Foundation property IDs, see “OPC DA Server Item Properties” on page B-2.

Using OPC Data Access Objects

To interact with an OPC server, you need to create toolbox objects. You create an OPC Data Access Client (`opcda` client) object to provide a connection to a particular OPC server. You then create one or more Data Access Groups (`dagroup` objects) to control sets of Data Access Items (`daitem` objects), which represent links to server items. OPC Data Access objects are described in more detail in “Toolbox Object Hierarchy for the Data Access Standard” on page 6-2.

- “Create OPC Data Access Objects” on page 6-2
- “Configure OPC Data Access Object Properties” on page 6-13
- “Delete Objects” on page 6-18
- “Save and Load Objects” on page 6-20

Create OPC Data Access Objects

In this section...

"Overview to Objects" on page 6-2
"Toolbox Object Hierarchy for the Data Access Standard" on page 6-2
"How Toolbox OPC Objects Relate to OPC DA Servers" on page 6-3
"Create Data Access Group Objects" on page 6-4
"Create Data Access Item Objects" on page 6-6
"Build an Object Hierarchy with a Disconnected Client" on page 6-7
"Create OPC Data Access Object Vectors" on page 6-8
"Work with Public Groups" on page 6-10

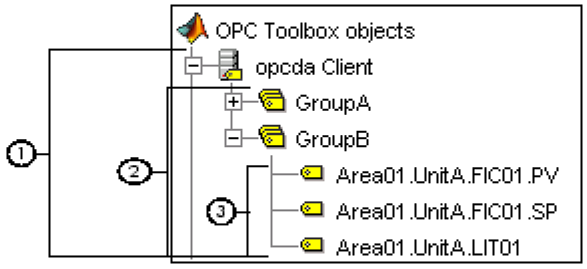
Overview to Objects

The first step in interacting with an OPC server from MATLAB is to establish a connection with the OPC server. You create `opcda` client objects to control the connection between an OPC server and the toolbox. Then you create `dagroup` objects to manage sets of `daitem` objects, and then you create the `daitem` objects themselves, which represent server items. A server item corresponds to a physical device or to a storage location in a SCADA system or DCS.

You must create the toolbox objects in the order described above. "Connect to OPC Data Access Servers" on page 5-4 describes how to create an `opcda` client object. This section discusses how to create and configure `dagroup` and `daitem` objects.

Toolbox Object Hierarchy for the Data Access Standard

OPC DA access in MATLAB is implemented using three basic objects, designed to help you manage connections to servers and collections of server items. The three objects are arranged in a specific hierarchy, shown in the following figure.



1 **OPC Data Access Client objects** (`opcda` client objects) represent a specific OPC client instance that can communicate with only one server. You define the server using the `Host` and `ServerID` properties. The `Host` property defines the computer on which the server is installed. The `ServerID` property defines the Program ID (*ProgID*) of the server, created when the server was installed on that host. The `opcda` client object acts as a container for multiple group objects, and manages the connection to the server, communication with the server, and server name space browsing.

2 **Data Access Group objects** (`dagroup` objects) represent containers for one or more server items (data points on the server.) A `dagroup` object manages how often the items in the group

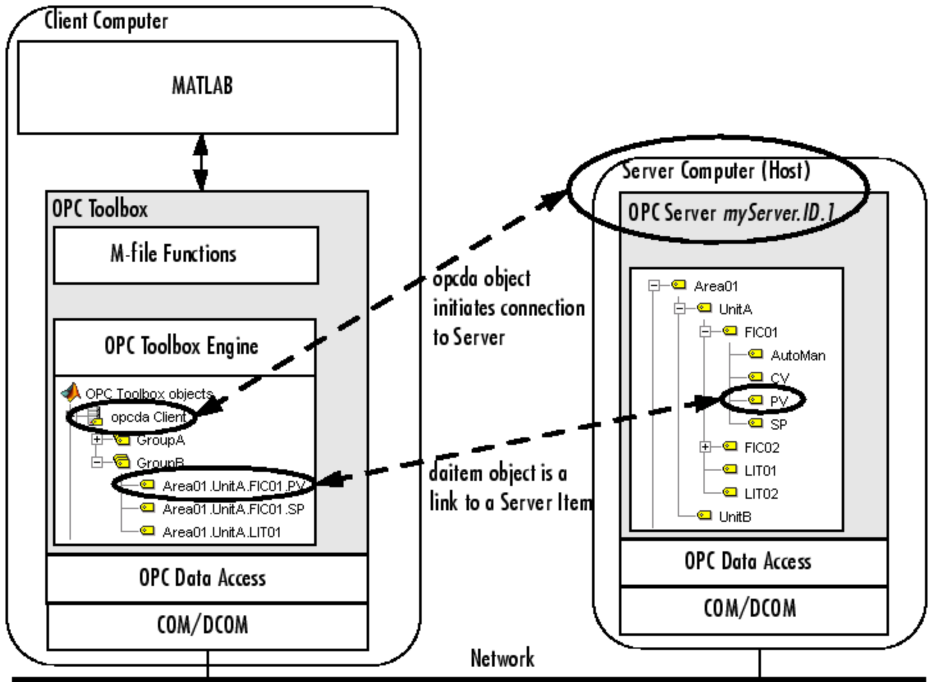
must be read, whether historical item information must be stored, and also manages creation and deletion of items. Groups cannot exist without an `opcda` client object. You create `dagroup` objects using the `addgroup` function of an `opcda` client object.

3 **Data Access Item objects** (`daitem` objects) represent server items. Items are defined by an *item ID*, which uniquely defines that server item in the server's name space. A `daitem` object has a `Value`, a `Quality`, and a `TimeStamp`, representing the information collected by the server from an instrument or data point in a SCADA system. The `Value`, `Quality`, and `TimeStamp` properties represent the information known to the server when the server was last asked to access information from that instrument.

A `dagroup` object can only exist “within” an `opcda` client object. Similarly, a `daitem` object can only exist within a `dagroup` object. You create `dagroup` objects using the `addgroup` method of an `opcda` client object. You create `daitem` objects using the `additem` method of the `dagroup` object.

How Toolbox OPC Objects Relate to OPC DA Servers

Industrial Communication Toolbox uses objects to define the server that the client must connect to, and the arrangement of items in groups. The following figure shows the relationship between the OPC Data Access objects and an OPC server.



The `opcda` client object establishes the connection between MATLAB and the OPC server, using OPC Data Access Specification standards. The standards are based on Microsoft COM/DCOM interoperability standards.

The `daitem` objects represent specific server items. Note that a client typically requires only a subset of the entire name space of a server in order to operate effectively. In the figure above, only the `PV` and `SP` items of `FIC01`, and the `LIT01` item, are required for that particular group. Another group may only contain a single `daitem` object, representing a single server item.

Note The dagroup object has no equivalent on the OPC server. However, the server keeps a record of each group that a client has created, and uses that group name to communicate to the client information about the items in that group.

Create Data Access Group Objects

Once you have created an opcda client object, you can add groups to the client. A dagroup object manages multiple daitem objects. Using a dagroup object, you can read data from all items in that group in one action, write data to the items in the group, define actions to take when any of the items in that group change value, or log data for all the items in that group for analysis and processing.

To create a dagroup object, you use the addgroup function, specifying the opcda client object that you want to add the group to, and an optional group name. See “Specify a Group Name” on page 6-4 for rules on defining your own group name.

The example below creates an opcda client object, connects that object to the server, and adds two groups to the client. The first group is automatically named by the server, and the second group is given a specified name.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp1 = addgroup(da);
grp2 = addgroup(da,'MyGroup');
```

Specify a Group Name

Each group created under a specific client object must have a unique name. This allows the OPC server to uniquely identify the group when a client makes a server request using that group. The name can be any nonempty character vector.

You do not need to specify a group name for each group that you add to a client. If you do not specify a name, the OPC server will automatically assign a group name for you. Each OPC server defines different rules for automatic naming of groups.

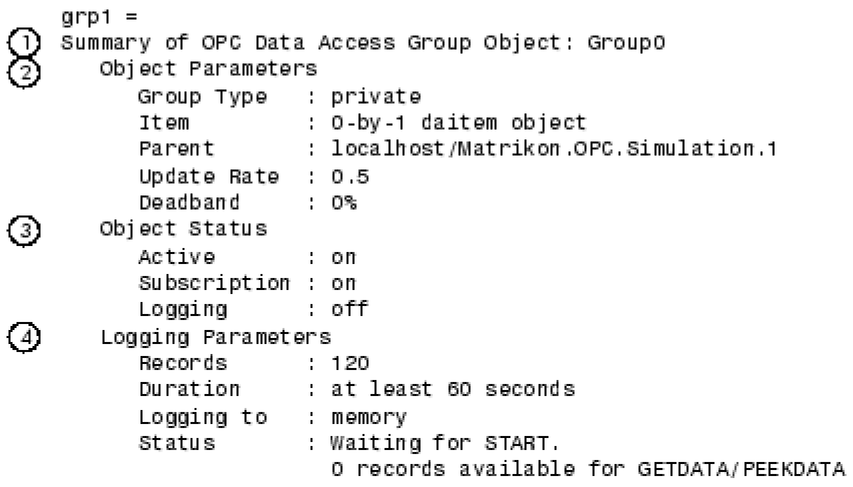
If you attempt to create a group with the same name as a group already created for that client, an error will be generated.

See “Delete Objects” on page 6-18 for information about how groups are automatically named when you create groups with a disconnected client.

View a Summary of a Group Object

To view a summary of the characteristics of the dagroup object you created, enter the variable name you assigned to the object at the command prompt. For example, this is the summary for the object grp1.

```
grp1
```



```
    grp1 =
①   Summary of OPC Data Access Group Object: Group0
②      Object Parameters
          Group Type   : private
          Item         : 0-by-1 daitem object
          Parent       : localhost/Matrikon.OPC.Simulation.1
          Update Rate  : 0.5
          Deadband     : 0%
③      Object Status
          Active       : on
          Subscription : on
          Logging      : off
④      Logging Parameters
          Records      : 120
          Duration     : at least 60 seconds
          Logging to   : memory
          Status       : Waiting for START.
                         0 records available for GETDATA/PEEKDATA
```

The items in this list correspond to the numbered elements in the object summary:

1 The title of the `Summary` includes the name of the `dagroup` object. In the example, this is the server-assigned name `Group0`.

2 The `Object Parameters` section lists the values of key `dagroup` object properties. These properties describe the type of group, the `daitem` objects associated with the group, the name of the group's parent `opcda` client object, and properties that control how the server updates item information for this group. In the example, any items created in this group will be updated at half-second intervals, with a deadband of 0%. For information on how the server updates item information, see “Data Change Events and Subscription” on page 7-8.

3 The `Object Status` section lists the current state of the object. A `dagroup` object can be in one of several states:

- The `Active` state defines whether any operation on the group applies to the item.
- The `Subscription` state defines whether changes in the item's value or quality will produce a data change event. See “Data Change Events and Subscription” on page 7-8 for more information about the Subscription property.
- The `Logging` state describes whether the group is logging or not. See “Log OPC Server Data” on page 7-11 for information on how to log data.

4 The `Logging Parameters` section describes the values of the logging properties for that group. Logging properties control how the `dagroup` object logs data, including the duration of the logging task and the destination of logged data. See “Log OPC Server Data” on page 7-11 for information on logging data using `dagroup` objects.

Use a Group Object

A `dagroup` object with no items does not perform any useful functions. Once you have added items to a group, you can use the group to

- Read data from, and write data to, the OPC server. See “Read and Write Data on OPC DA Server” on page 7-2 for more information.
- Control how an OPC server notifies MATLAB about changes in any item associated with a `dagroup` object. See “Data Change Events and Subscription” on page 7-8 for more information.
- Log data from all items in that group, for later processing and analysis. “Log OPC Server Data” on page 7-11 describes how to control logging.

Create Data Access Item Objects

A `dagroup` object provides a container for collecting one or more `daitem` objects. A `daitem` object provides a link to a specific server item. The `daitem` object defines how you want to retrieve and store the client-side value of the server item, and also stores the last data retrieved from the server for that server item. You can use a `daitem` object to read data from the server for that server item, or to write values to that server item on the server.

You create a `daitem` object using the `additem` function, specifying the `dagroup` object to which the item must be added and the fully qualified item ID of the server item. You can obtain a list of the fully qualified item IDs for all server items using the `serveritems` function.

The example below builds on the example in “Create Data Access Group Objects” on page 6-4 by adding a `daitem` object to the first group created in that example. The server item associated with this item is called `'Random.Real8'`.

```
itm1 = additem(grp1,'Random.Real8');
```

Specify a Local Data Type for the Item

When you create a `daitem` object, you create an object that stores the value of the server item locally on the client. You can specify that the local storage data type be different from the server storage data type. For example, you can specify that a value stored on the server as an integer be stored in MATLAB as a double-precision floating-point value, because you know that you will be performing double-precision calculations with that item's value.

Although it is possible to modify the data type of the item after it is created, you can also create an item with a specific data type by specifying the data type as the third parameter to the `additem` function. The data type specification must be a character vector describing that data type. Valid OPC data types are any MATLAB numeric data type, plus `'char'`, and `'logical'`. See “Work with Different Data Types” on page 8-13 for more information on supported data types.

The example below adds another item to the group `grp1` created by the example in “Create Data Access Group Objects” on page 6-4. The item ID is `'Random.UInt2'`, which is stored on the server as an unsigned 16-bit integer. By specifying the data type as `'double'`, the value will be returned to MATLAB and stored locally as a double-precision floating-point number.

```
itm2 = additem(grp1,'Random.UInt2','double');
```

Note The conversion process from the server's data type to the item's data type is performed by the server, using Microsoft COM Variant conversion rules. If you attempt to convert a value to a data type that does not have that value's range, the OPC server will return an error when attempting to update the value of that item. You should then change the data type to one that has the same or larger range than the server item's data type. See “Work with Different Data Types” on page 8-13 for more information.

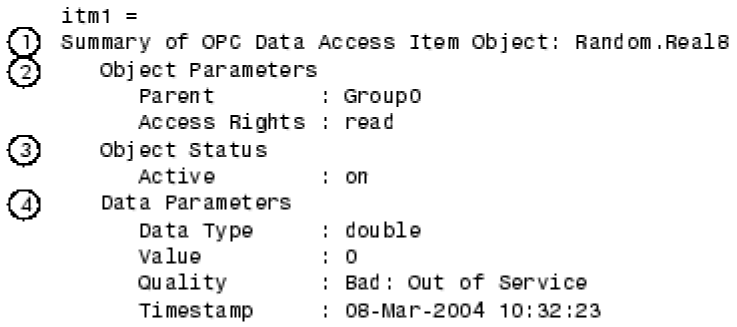
Specify the Active Status of an Item Object

You can optionally specify the `Active` status of an `daitem` object by passing a character vector as the fourth parameter to the `additem` function. The `Active` status can be `'on'` or `'off'`. An item with an `Active` status of `'off'` behaves as if the item was never created: No server updates of the item's value will take place, and a read or write with that item will fail. You use the `Active` status to temporarily disable an item without deleting that item from MATLAB. For more information on the `Active` status, see the reference page for the Active property.

View a Summary of the Item Object

To view a summary of the characteristics of the `daitem` object you created, enter the variable name you assigned to the object at the command prompt. For example, this is the summary for the object `itm1`.

```
itm1
```



```
itm1 =
Summary of OPC Data Access Item Object: Random.Real8
   Object Parameters
      Parent        : Group0
      Access Rights : read
   Object Status
      Active        : on
   Data Parameters
      Data Type     : double
      Value         : 0
      Quality       : Bad: Out of Service
      Timestamp     : 08-Mar-2004 10:32:23
```

The items in this list correspond to the numbered elements in the object summary:

1. The title of the `Summary` includes the fully qualified item ID of the item. In the example, the item is associated with the `'Random.Real8'` server item.
2. The `Object Parameters` section lists the values of key `daitem` object properties. These properties describe the name of the item's `Parent` group, and the `Access Rights` advertised by the server.
3. The `Object Status` section lists the Active state of the object. The `Active` state defines whether any operation on the parent group applies to the item, and whether you want to be notified of any changes in the item's value.
4. The `Data Parameters` section lists the `Data Type` used by the `daitem` object to store the value, and the `Value`, `Quality`, and `TimeStamp` of the last value obtained from the server for this item. For more information on the `Value`, `Quality`, and `TimeStamp` of an item, see “OPC Data: Value, Quality, and TimeStamp” on page 8-2.

Use an Item Object

You create a `daitem` object to query the value of the associated server item, or to write values to that server item. You can write values to a single item, and read values from a single item, using the `daitem` object. For more information on reading and writing values, see “Read and Write Data on OPC DA Server” on page 7-2.

You can also use the parent `dagroup` object to read and write values for all of the `daitem` objects contained in that group, or to log changes in the item's value for a period of time. See “Log OPC Server Data” on page 7-11 for information on logging data.

Build an Object Hierarchy with a Disconnected Client

When you create objects with a connected client, the toolbox validates those objects with the OPC server before creating them on the client. For example, when adding a group to the client using the `addgroup` function, the validation process ensures that no other group with the same name exists on the server, and that the server will accept the group. When adding an item, the item ID is verified to be a valid server item.

Occasionally you might want to build up a toolbox object hierarchy without connecting to the server. For example, you might be off site and want to configure a logging task for use on the following day, rather than wait to configure the objects for that task when you are on site.

You can configure an entire OPC object hierarchy without connecting to the server. However, without a connection to the server, the toolbox cannot validate the created objects with that server. Instead, the toolbox performs some basic validation on the objects you create, and revalidates those objects with the server when you connect.

When you create toolbox objects with a disconnected client, the following validation is performed:

- When adding a group using the `addgroup` function, if you do not specify a name, the toolbox assigns a unique name `'group`*`N`*`'`, where *`N`* is the lowest integer that ensures that the group name is unique. For example, the first group created will be `'group1'`, then `'group2'`, and so on.
- When you specify a group name when using the `addgroup` function, an error is generated if a group with the same name already exists.
- When adding an item to a group using the `additem` function, an error is generated only if an item with the same name already exists in that group. No other checking is performed on the item.
- When adding an item to a group, if you do not specify a data type for that item, the data type is set to `'unknown'`. When you connect to the server, the data type will be changed to the server item's `CanonicalDataType`.

Despite all of the checks described above, the server might not accept all objects created on a disconnected client when that client is connected to the server using the `connect` function. For example, an item's item ID might not be valid for that server, or a group name might not be valid for that server. When you connect a client to the server using `connect`, any objects that the server rejects will be deleted from the object hierarchy, and a warning will be generated. In this way, all objects on a connected client are guaranteed to have been accepted by the server.

Create OPC Data Access Object Vectors

An object vector is a single variable in the MATLAB workspace containing a reference to more than one object. For example, all the groups added to an `opcda` client object are stored in the client Group property. The `Group` property contains a `dagroup` object vector that represents all groups in that client. Similarly, a `dagroup` object has an `Item` property that contains a reference to every `daitem` object created in the group.

You can construct vectors using any of the standard concatenation techniques available in MATLAB. However, Industrial Communication Toolbox imposes some limitations on the construction of object vectors:

- Objects must be the same class. For example, you can concatenate two `dagroup` objects, but you cannot concatenate a `dagroup` object with a `daitem` object.
- Group and item objects must have the same parent.
- One of the dimensions of the resulting array must be scalar. You can create a column vector (*m*-by-1 objects) or a row vector (1-by-*n* objects), but not an *m*-by-*n* matrix.
- Industrial Communication Toolbox does not fill in missing elements in a vector. Instead, an error is generated. For example, you cannot assign a scalar object at the 4th index to a scalar object.

The following sections discuss how to create and use toolbox object vectors:

- “Construct Object Vectors” on page 6-9 describes how to create object vectors.
- “Display a Summary of Object Vectors” on page 6-9 describes how object vectors are displayed at the command line.
- “Use Object Vectors” on page 6-10 describes how you can use OPC object vectors.

Construct Object Vectors

You can construct an object vector using any of the following techniques:

- Using concatenation of lists of individual object variables
- Using indexed assignment
- Using object properties to retrieve object vectors

Create Object Vectors Using Concatenation

To construct an OPC Data Access object vector using concatenation, use the normal MATLAB syntax for concatenation. Create a list of all objects you want to create, and surround that list with square brackets (`[]`). Separate each element of the object vector by either a comma (`,`) to create a row vector, or a semicolon (`;`) to create a column vector.

The following example creates three fictitious `opcda` client objects, and concatenates them into a row vector.

```
da1 = opcda('Host1','Dummy.Server.1');
da2 = opcda('Host2','Dummy.Server.2');
da3 = opcda('Host3','Dummy.Server.3');
dav = [da1, da2, da3];
```

Create Object Vectors Using Indexed Assignment

Indexed assignment refers to creating vectors by assigning elements to specific indices in the vector. The following example constructs the same three-element `opcda` client object vector as in the previous example, using indexed assignment.

```
dav(1) = opcda('Host1','Dummy.Server.1');
dav(2) = opcda('Host2','Dummy.Server.2');
dav(3) = opcda('Host3','Dummy.Server.3');
```

Create an Object Vector Using Object Properties

You may obtain an object vector if you assign the `Group` property of a `opcda` client object, or the `Item` property of a `dagroup` object, to a variable. If the client has more than one group, or the group has more than one item, the resulting property is an object vector.

For information on obtaining object properties, see “View the Value of a Particular Property” on page 6-14.

Display a Summary of Object Vectors

To view a summary of an object vector, type the name of the object vector at the command prompt. For example, this is the summary of the client vector `dav`.

```
dav
```

```
OPC Data Access Object Array:

Index:  Status:         Name:
1       disconnected    Host1/Dummy.Server.1
2       disconnected    Host2/Dummy.Server.2
3       disconnected    Host3/Dummy.Server.3
```

The summary information for each OPC Data Access object class is different. However, the basic display is similar.

Use Object Vectors

You use object vectors just as you would a normal object variable. The function you call with the object vector simply gets applied to all objects in the vector. For example, passing the client vector `dav` to the connect function connects each object in the vector to its OPC server.

Note Some functions do not accept object vectors as arguments. If you attempt to use an object vector with a function that does not accept object vectors, an error is generated. Consult the relevant function reference page for information on whether a function supports object vectors.

If you need to extract elements of an object vector, use standard MATLAB indexing notation. For example, the following example extracts the second element from the client vector `dav`.

```
dax = dav(2);
```

Work with Public Groups

The OPC Data Access Specification provides a mechanism for sharing group configuration amongst many clients. Normally, a client has *private* access to a group; no other client connected to the same server can see that group, and the items configured in that group. However, a client can define a group as *public*, allowing other clients connected to the same server to gain access to that group.

Note The OPC Data Access Specification defines the support for public groups as optional. Consequently, some OPC servers will not support public groups.

A public group differs from a private group in the following ways:

- Once a group is defined as public, you cannot add items to that group, nor remove items from the group. This restriction ensures that every client using that public group has access to the same items, and does not need to worry about items being added or removed from that group. You should ensure that a group's items are correct before making that group public.
- Each client that accesses the public group is able to set its own group properties, such as the `UpdateRate`, `DeadbandPercent`, `Active`, and `Subscription` properties. For example, one client can define an `UpdateRate` of `10` seconds for a public group, while another client specifies the `UpdateRate` as `2` seconds.
- Each public group defined on a server must have a unique name. If you attempt to create a public group with a name that is the same as a public group on the server, an error is generated.

- A single client cannot have a public group and a private group with the same name. For example, you cannot connect to a public group named `'LogGroup'` and then create a private group called `'LogGroup'`.

You can define and publish your own public groups or connect to existing public groups. You an also request that public groups be removed from an OPC server. The following sections illustrate how you can work with public groups:

- “Define a New Public Group” on page 6-11 describes how you can create new public groups.
- “Connect to an Existing Public Group” on page 6-11 describes how you can utilise a public group that is already defined on the server.
- “Remove Public Groups from the Server” on page 6-12 describes how you can remove public groups from an OPC server.

Define a New Public Group

You define a new public group by creating a private group in the normal way (described in “Create Data Access Group Objects” on page 6-4) and then converting that private group into a public group.

You use the `makepublic` function to convert a private group into a public group. The only argument to the `makepublic` function is the group object that you want to convert to a public group.

The following example creates a private group, with specific items in that group. The group is then converted into a public group.

```
da = opcda('localhost','My.Server.1');
grp = addgroup(da,'PublicGrpExample');
itms = additem(grp,{'Item.ID.1','Item.ID.2'});
makepublic(grp);
```

You can check the group type using the GroupType property.

```
grp.GroupType
```

```
public
```

Connect to an Existing Public Group

In addition to creating new public groups, you can also create a connection to an existing public group on the server. To obtain a list of available public groups on a server, you use the `opcserverinfo` function, passing the client object that is connected to the server as the argument. The returned structure includes a field called `'PublicGroups'`, containing a cell array of public groups defined on that server. If the `'PublicGroups'` field is empty, then you should check the `'SupportedInterfaces'` field to ensure that the server supports public groups. A server that supports public groups will implement the `IOPCServerPublicGroups` interface.

Once you have a list of available public groups, you can create a connection to that group using the `addgroup` function, passing it the client that is connected to the server containing the public group, the name of the public group, and the `'public'` group type specifier.

Note You cannot create a connection to an existing public group if your client object is disconnected from the server.

The following example connects to a public group named `'PublicTrends'` on the server with program ID `'My.Server.1'`.

```
da = opcda('localhost','My.Server.1');
connect(da);
pubGrp = addgroup(da,'PublicTrends','public');
```

When you connect to a public group, the items in that group are automatically created for you.

```
itm = pubGrp.Items
```

```
itm =

   OPC Item Object Array:

   Index:  DataType:  Active:  ItemID:
   1       double     on       item.id.1
   2       uint16     on       item.id.2
   3       double     on       item.id.3
```

You cannot add items to or remove items from a public group. However, you can still modify the update rate of the group, the dead band percent, and the active and subscription status of the group, and you can use the group to read, write, or log data as you would for a private group.

When you have finished using a public group, you can use the `delete` function to remove that group from your client object. Deleting the group from the client does not remove the public group from the server; other clients might require that group after you have finished with it. Instead, deleting the group from the client indicates to the server that you are no longer interested in the group.

Remove Public Groups from the Server

You can request that a public group be removed from a server using the `removepublicgroup` function, passing the client object that is connected to the server and the name of the public group to remove.

Caution The OPC Data Access Specification does not provide any security mechanism for removing public groups; any client can request that a public group be removed. You should use this function with extreme caution!

If any clients are currently connected to that group, the server will issue a warning stating that the group will be removed when all clients have finished using the group.

Configure OPC Data Access Object Properties

In this section...

"Purpose of Object Properties" on page 6-13

"View the Values of Object Properties" on page 6-13

"View the Value of a Particular Property" on page 6-14

"Get Information About Object Properties" on page 6-15

"Set the Value of an Object Property" on page 6-15

"View a List of All Settable Object Properties" on page 6-16

Purpose of Object Properties

All OPC Data Access objects support properties that enable you to control characteristics of the object:

- The `opcda` client object properties control aspects of the connection to the OPC server, and event information obtained from the server. For example, you can use the `Timeout` property to define how long to wait for the server to respond to a request from the client.
- The `dagroup` object properties control aspects of the collection of items contained within that group, including all logging properties. For example, the `UpdateRate` property defines how often the items in the group must be checked for value changes, as well as the rate at which data will be sent from the server during a logging session.
- The `daitem` object properties control aspects of a single server item. For example, you use the `DataType` property to define the data type that the server must use to send values of that server item to the toolbox.

For all three toolbox objects, you can use the same toolbox functions to

- View a list of all the properties supported by the object, with their current values
- View the value of a particular property
- Get information about a property
- Set the value of a property

View the Values of Object Properties

To view all the properties of an OPC Data Access object, with their current values, use the `get` function.

If you do not specify a return value, the `get` function displays the object properties in categories that group similar properties together. Use the display form of the `get` function to view the value of all properties for the toolbox object.

This example uses the `get` function to display a list of all the properties of the OPC `dagroup` object `grp`.

```
get(grp)
```

```
  General Settings:
    DeadbandPercent = 0
    GroupType = private
    Item = []
    Name = group1
    Parent = [1x1 opcda]
    Tag = 
    TimeBias = 0
    Type = dagroup
    UpdateRate = 0.5000
    UserData = []

  Callback Function Settings:
    CancelAsyncFcn = @opccallback
    DataChangeFcn = []
    ReadAsyncFcn = @opccallback
    RecordsAcquiredFcn = []
    RecordsAcquiredFcnCount = 20
    StartFcn = []
    StopFcn = []
    WriteAsyncFcn = @opccallback

  Subscription and Logging Settings:
    Active = on
    LogFileName = opcdatalog.olf
    Logging = off
    LoggingMode = memory
    LogToDiskMode = index
    RecordsAcquired = 0
    RecordsAvailable = 0
    RecordsToAcquire = 120
    Subscription = on
```

View the Value of a Particular Property

To view the value of a particular property of an OPC Data Access object, use the `get` function, specifying the name of the property as an argument. You can also access the value of the property as you would a field in a MATLAB structure.

This example uses the `get` function to retrieve the value of the `Subscription` property for the `dagroup` object.

```
get(grp,'Subscription')
```

```
ans =

on
```

This example illustrates how to access the same property by referencing the object as if it were a MATLAB structure.

```
grp.Subscription
```

```
ans =

on
```

Get Information About Object Properties

To get information about a particular property, use the`propinfo` or `opchelp` function.

The `propinfo` function returns a structure that contains information about the property, such as its data type, default value, and a list of all possible values if the property supports such a list. This example uses `propinfo` to get information about the `LoggingMode` property.

```
propinfo(grp,'LoggingMode')
```

```
ans =

               Type: 'string'
         Constraint: 'enum'
    ConstraintValue: {'memory'  'disk'  'disk&memory'}
       DefaultValue: 'memory'
           ReadOnly: 'whileLogging'
```

The `opchelp` function returns reference information about the property with a complete description. This example uses `opchelp` to get information about the `LoggingMode` property.

```
opchelp(grp,'LoggingMode')
```

Set the Value of an Object Property

To set the value of a particular property of an OPC Data Access object, use the `set` function, specifying the name of the property as an argument. You can also assign the value to the property as you would a field in a MATLAB structure.

Note Because some properties are read-only, only a subset of the toolbox object properties can be set. Use the property reference pages or the `propinfo` function to determine if a property is read-only.

This example uses the `set` function to set the value of the `LoggingMode` property.

```
set(grp,'LoggingMode','disk&memory')
```

To verify the new value of the property, use the `get` function.

```
get(grp,'LoggingMode')
```

```
ans =

disk&memory
```

This example sets and views the value of a property by using dot-notation.

```
grp.LoggingMode = 'disk';
grp.LoggingMode
```

```
ans =

disk
```

View a List of All Settable Object Properties

To view a list of all the properties of a toolbox object that can be set, use the `set` function.

```
set(grp)
```

```
  General Settings:
    DeadbandPercent
    Name
    Tag
    TimeBias
    UpdateRate
    UserData

  Callback Function Settings:
    CancelAsyncFcn: character vector -or- function handle -or- cell array
    DataChangeFcn: character vector -or- function handle -or- cell array
    ReadAsyncFcn: character vector -or- function handle -or- cell array
    RecordsAcquiredFcn: character vector -or- function handle -or- cell array
    RecordsAcquiredFcnCount
    StartFcn: character vector -or- function handle -or- cell array
    StopFcn: character vector -or- function handle -or- cell array
    WriteAsyncFcn: character vector -or- function handle -or- cell array

  Subscription and Logging Settings:
    Active: [ {on} | off ]
    LogFileName
    LoggingMode: [ {memory} | disk | disk&memory ]
    LogToDiskMode: [ {index} | append | overwrite ]
    RecordsToAcquire
    Subscription: [ {on} | off ]
```

When using the `set` function to display a list of settable properties, all properties that have a predefined set of acceptable values list those values after the property. The default value is enclosed in curly braces (`{}`). For example, from the display shown above, you can set the `Subscription` property for a `dagroup` object to `'on'` or `'off'`, with the default value being `'on'`. You can set the `LogFileName` property to any value.

Special Read-Only Modes

Some OPC Data Access object properties change their read-only status, depending on the state of an object (defined by another property of that object, or the parent of that object). The toolbox uses two special read-only modes:

- `'whileConnected'`: These properties cannot be changed while the client is connected to the OPC server. For example, the client's Host property is read-only while connected.
- `'whileLogging'`: These properties cannot be changed while the `dagroup` object is logging. For example, the LoggingMode property is read-only while logging. For more information on logging, see “Log OPC Server Data” on page 7-11.
- `'whilePublic'`: These properties cannot be changed because the group is a public group. For more information on public groups, see “Work with Public Groups” on page 6-10.

Note Properties that modify their read-only state are always displayed when using `set` to display settable properties, even when they cannot be changed because of the state of the object.

To determine if a property has a modifiable read-only state, use the `propinfo` function.

See Also

Properties
opcda Object Properties Properties | dagroup Object Properies Properties | daitem Object Properties Properties

Delete Objects

When you finish using your OPC Data Access objects, use the `delete` function to remove them from memory. After deleting them, clear the variables that reference the objects from the MATLAB workspace by using the `clear` function.

Note When you delete an `opcda` client object, all the group and item objects associated with the `opcda` client object are also deleted. Similarly, when you delete a `dagroup` object, all `daitem` objects associated with that `dagroup` object are deleted.

To illustrate the deletion process, this example creates several `opcda` client objects and then deletes them.

Step 1: Create several clients

This example creates several `opcda` client objects using fictitious host and server ID properties.

```
da1 = opcda('Host1','Dummy.Server.1');
da2 = opcda('Host2','Dummy.Server.2');
da3 = opcda('Host3','Dummy.Server.3');
```

Step 2: Delete clients

Always remove toolbox objects from memory, and the variables that reference them, when you no longer need them.

You can delete toolbox objects using the `delete` function.

```
delete(da1)
delete(da2)
delete(da3)
```

Note that the variables associated with the objects remain in the workspace.

```
whos
```

```
  Name      Size                    Bytes  Class

  da1       1x1                       636  opcda object
  da2       1x1                       636  opcda object
  da3       1x1                       636  opcda object
```

These variables are not valid OPC Data Access objects.

```
isvalid(da1)
```

```
ans =
    0
```

To remove these variables from the workspace, use the `clear` command.

Note You can delete toolbox object vectors using the `delete` function. You can also delete individual elements of a toolbox object vector.

See Also

Functions
`clear` | `delete` | `opcreset` | `isvalid`

Save and Load Objects

Using the `save` command, you can save an OPC Data Access object to a MAT-file, just as you would any workspace variable. This example saves the `dagroup` object `grp` to the MAT-file `myopc.mat`.

```
save myopc grp
```

When you save a toolbox object, all the toolbox objects in that object hierarchy are also saved. For example, if you save a `dagroup` object, the client, all groups associated with that client and all items created in those groups are saved along with the `dagroup` object. However, only those objects you elect to save will be created in the MATLAB workspace. Other objects will be created with no reference to them in the workspace. To obtain a reference to an existing OPC Data Access object, use the `opcfind` function.

To load a toolbox object that was saved to a MAT-file into the MATLAB workspace, use the `load` command. For example, to load `grp` from MAT-file `myopc.mat`, use

```
load myopc
```

Note The values of read-only properties are not saved. When you load a toolbox object into the MATLAB workspace, read-only properties revert back to their default values. To determine if a property is read-only, use the `propinfo` function.

See Also

Functions
`copyobj` | `opcfind` | `save` | `load`

Reading, Writing, and Logging OPC Data

The core of any OPC application is the exchange of data between the MATLAB workspace and one or more OPC servers. You create and configure toolbox objects to support the reading, writing, and data logging functions that you require for your application.

You can exchange data with an OPC server in a number of ways. You can read and write data from the MATLAB command line or other MATLAB functions. You can configure toolbox OPC objects to automatically run MATLAB code when the server notifies the objects that data has changed on the server. You can also log changes in OPC server data to a disk file or to memory for further analysis.

This chapter provides information on how to exchange data with an OPC server.

- “Read and Write Data on OPC DA Server” on page 7-2
- “Data Change Events and Subscription” on page 7-8
- “Log OPC Server Data” on page 7-11

Read and Write Data on OPC DA Server

In this section...

"Introduction to Reading and Writing" on page 7-2
"Read Data from an Item" on page 7-2
"Write Data to an Item" on page 7-4
"Read and Write Multiple Values" on page 7-6

Introduction to Reading and Writing

You can exchange data with the OPC DA server using individual items, or using the `dagroup` object to perform the operation on multiple items. The reading and writing operation can be performed *synchronously*, so that your MATLAB session waits for the operation to complete, or *asynchronously*, allowing your MATLAB session to continue processing while the operation takes place in the background.

Read Data from an Item

You can read data from any item that is associated with a connected client. When you perform the read operation on an item, the server will return information about the server item associated with that item ID. The read operation can be performed synchronously or asynchronously:

- "Use Synchronous Read Operations" on page 7-2 describes how to perform synchronous read operations. Synchronous read operations can request data from the server's cache, or directly from the device.
- "Use Asynchronous Read Operations" on page 7-4 describes how to perform asynchronous read operations.

Use Synchronous Read Operations

A *synchronous* read operation means that MATLAB waits for the server to return data from a read request before continuing processing. The data returned by the server can come from the server's cache, or you can request that the server read values from the device that the server item refers to.

You use the `read` function to perform synchronous read operations, passing the `daitem` object associated with the server item you want to read. If the read operation is successful, the data is returned in a structure containing information about the read operation, including the value of the server item, the quality of that value, and the time that the server obtained that value. The item's `Value`, `Quality` and `Timestamp` properties are also updated to reflect the values obtained from the read operation.

The following example creates an `opcda` client object and configures a group with one item, `'Random.Real8'`. A synchronous read operation is then performed on the item.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da);
itm1 = additem(grp,'Random.Real8');
r = read(itm1)
```

```
r =
       ItemID: 'Random.Real8'
        Value: 4.3252e+003
      Quality: 'Good: Non-specific'
    TimeStamp: [2004 3 2 9 50 26.6710]
        Error: ''
```

Specify the Source of the Read Operation

By default, a synchronous read operation will return data from the OPC server's cache. By reading from the cache, you do not have to wait for a possibly slow device to provide data to the server. You can specify the source of the synchronous read operation as the second parameter to the `read` function. If the source is specified as `'device'`, the server will read a value from the device, and return that value to you (as well as updating the server cache with that value).

Note Reading from the device may be slow. If the read operation generates a time-out error, you may need to increase the value of the `Timeout` property of the `opcda` client object associated with the group or item in order to support synchronous reads from the device.

The following example reads data from the device associated with `itm1`.

```
r = read(itm1,'device')
```

```
r =

       ItemID: 'Random.Real8'
        Value: 9.1297e+003
      Quality: 'Good: Non-specific'
    TimeStamp: [2004 3 2 10 8 20.2650]
        Error: ''
```

Read from the Cache with Inactive Items

In order to reduce communication traffic and speed up data access, OPC servers do not store all server item values in their cache. Only those server items that are *active* will be stored in the server cache. Therefore, synchronous read operations from the cache on an inactive item will return data that may not correspond to the current device value. If you attempt to read data from an inactive item using the `read` function, and do not specify `'device'` as the source, the `Quality` will be set to `'Bad: Out of Service'`.

You control the active status of an item using the Active property.

The following example sets the `Active` property of the item to `'off'` and attempts to read from the cache.

```
itm1.Active = 'off';
r = read(itm1)
```

```
Warning: One or more items is inactive.
(Type "warning off opc:read:iteminactive" to suppress this
warning.)

r =

       ItemID: 'Random.Real8'
        Value: 8.4278e+003
      Quality: 'Bad: Out of Service'
```

```
    TimeStamp: [2004 3 2 10 17 19.9370]
        Error: ''
```

Use Asynchronous Read Operations

An *asynchronous* read operation creates a request to read data, and then sends that request to the server. Once the request has been accepted, MATLAB continues processing the next instruction without waiting to receive any values from the server. When the data is ready to be returned, the server sends the data back to MATLAB by generating a *read async event*. MATLAB will handle that event as soon as it is able to perform that task.

Asynchronous read operations always return data from the device.

By using an asynchronous read operation, you can continue performing tasks in MATLAB while the value is being read from the device, and then process the returned value when the server is able to provide it back to MATLAB.

You perform asynchronous read operations using the `readasync` function, passing the `daitem` object that you want to read from. If successful, the function will return a *transaction ID*, a unique identifier for that asynchronous transaction. You can use that transaction ID to identify the read operation when it is returned through the read async event.

When an asynchronous read operation is processed in MATLAB, the item's `Value`, `Quality` and `Timestamp` properties are also updated to reflect the values obtained from the asynchronous read operation.

The following example of using an asynchronous read operation uses the default callback for a read async event. The default callback is set to the `opccallback` function, which displays information about the event in the command line.

```
tid = readasync(itm1)
```

```
tid =

     3
```

The transaction ID for this operation is 3. A little while later, the default callback function displays the following information at the command line.

```
OPC ReadAsync event occurred at local time 10:44:49
    Transaction ID: 3
    Group Name: Group0
    1 items read.
```

You can change the read async event callback function by setting the `ReadAsyncFcn` property of the `dagroup` object.

Write Data to an Item

You can write data to individual items, or to groups of items. This section describes how to write data to individual items. See “Read and Write Multiple Values” on page 7-6 for information on using `dagroup` objects to write data to multiple items.

You can write data to an OPC server using a synchronous write operation, in which case MATLAB will wait for the server to acknowledge that the write operation succeeds, or using an asynchronous write operation, in which case MATLAB is free to continue performing other tasks while the write operation takes place. Because write operations always apply directly to the device, a synchronous write operation may take a significant amount of time, particularly if the device that you are writing to has a slow connection to the OPC server.

Use Synchronous Write Operations

You use the `write` function to perform synchronous write operations. The first argument is the `daitem` object that represents the server item you want to write to. The second argument is the value that you want to write to that server item. The `write` function does not return any results, but will generate an error if the write operation is not successful.

The following example creates an item with item ID `'Bucket Brigade.Real8'` and writes the value `10.34` to the item. The value is then read using a synchronous read operation.

```
itm2 = additem(grp,'Bucket Brigade.Real8');
write(itm2, 10.34)
r = read(itm2,'device')
```

You do not need to ensure that the data type of the value you are writing, and the data type of the `daitem` object, are the same. Industrial Communication Toolbox relies on the server to perform the conversion from the data type you provide, to the data type required for that server item. For information on how the toolbox handles different data types, see “Work with Different Data Types” on page 8-13.

Use Asynchronous Write Operations

An asynchronous write operation creates a request to write data, and then sends that request to the server. Once the request has been accepted, MATLAB continues processing the next instruction without waiting for the data to be written. When the write operation completes on the server, the server notifies MATLAB that the operation completed by generating a *write async event* containing information on whether the write operation succeeded, and an error message if applicable. MATLAB will handle that event as soon as it is able to perform that task.

You use the `writeasync` function to write values to the server asynchronously. The first argument is the `daitem` object that represents the server item you want to write to. The second argument is the value you want to write to that server item. The return value is the *transaction ID* of the operation. You can use the transaction ID to identify the write operation when it is returned through the write async event.

The following example uses asynchronous operations to write the value `57.8` to the item `'Bucket Brigade.Real8'` created earlier.

```
tid = writeasync(itm2, 57.8)
```

```
tid =

     4
```

A while later, the standard callback (`opccallback`) will display the results of the write operation to the command line.

```
OPC WriteAsync event occurred at local time 11:15:27
   Transaction ID: 4
   Group Name: Group0
   1 items written.
```

You can change the write async event callback function by setting the `WriteAsyncFcn` property of the `dagroup` object.

Read and Write Multiple Values

When you use the read and write operation on a single `daitem` object, you read or write a single value per transaction. Industrial Communication Toolbox allows you to perform one operation to read multiple item values, or to write multiple values. You can also use a `dagroup` object to read and write values using all items in the group, or you can perform read and write operations on item object vectors.

A `daitem` object vector is a single variable in the MATLAB workspace containing more than one `daitem` object. You can construct item vectors using any of the standard concatenation techniques available in MATLAB. See “Create OPC Data Access Object Vectors” on page 6-8 for information on creating and working with toolbox object vectors.

When you perform any read or write operation on a `dagroup` object, it is the equivalent of performing the operation on the `Item` property of that group, which is a `daitem` object vector representing all items that are contained within the `dagroup` object.

The following sections describe how to perform read and write operations on multiple items:

- “Read Multiple Values” on page 7-6 describes how to read multiple values from an item vector or `dagroup` object.
- “Write Multiple Values” on page 7-7 describes how to write multiple values to an item vector or `dagroup` object.
- “Error Handling for Multiple Item Read and Write Operations” on page 7-7 explains errors when performing read and write operations on multiple objects.

Read Multiple Values

The following sections describe how synchronous read operations and asynchronous read operations behave for multiple items.

Synchronous Read Operations

When you read multiple values using the `read` function, the returned value will be a structure array. Each element of the structure will contain the same fields. One of the fields is the item ID that the information in that element of the structure refers to.

The following example performs a synchronous read operation on the `dagroup` object created in the previous examples in this section.

```
r = read(grp)
```

```
r = 

2x1 struct array with fields:
    ItemID
    Value
    Quality
    TimeStamp
    Error
```

To display the first record in the structure array, use indexing into the structure.

```
r(1)

ans = 

       ItemID: 'Random.Real8'
        Value: 3.7068e+003
      Quality: 'Good: Non-specific'
    TimeStamp: [2004 3 2 11 49 52.5460]
        Error: ''
```

To display all values of a particular field, you can use the list generation syntax in MATLAB. Enclosing that list in a cell array groups the values into one variable.

```
{r.Value}

ans = 

  {3.7068e+003    10}
```

Asynchronous Read Operations

When you read multiple values using the `readasync` function, the return value is still a single transaction ID. The multiple values will be returned in the read async event structure passed to the `ReadAsyncFcn` callback. For information on the structure of the read async event, see "Event Types" on page 9-4.

Write Multiple Values

When you perform a write operation on multiple items you need to specify multiple values, one for each item you are writing to. These multiple values must be in a cell array, because the data types for each value might be different.

Note Even if you are using the same data type for every value being written to the `dagroup` object or `daitem` object vector, you must still use a cell array to specify the individual values. Use the `num2cell` function to convert numeric arrays to cell arrays.

The following example writes values to a `dagroup` object containing two items.

```
write(grp, {1.234, 5.43})
```

Error Handling for Multiple Item Read and Write Operations

When reading and writing with multiple items, an error generated by performing the operation on one item will not automatically generate an error in MATLAB. The following rules apply to reading and writing with multiple items:

- If all items fail the operation, an error will be generated. The error message will contain specific information for each item about why the item failed.
- If some items fail but some succeed, the operation does not error, but generates a warning, listing which items failed and the reason for failure.

Note that for asynchronous read and write operations, items may fail early (during the request for the operation) or late (when the information is returned from the server). If any items fail late, an error event will be generated in addition to the read async event or write async event.

Data Change Events and Subscription

In this section...

"Introduction to Data Change Events" on page 7-8
"Configure OPC Objects for Data Change Events" on page 7-8
"How Data Change Events are Processed" on page 7-9
"Customize the Data Change Event Response" on page 7-10

Introduction to Data Change Events

Using the `read` and `readasync` functions described in "Read Data from an Item" on page 7-2, you can obtain information about OPC server item values upon request. The OPC Data Access specification provides another mechanism for clients to get information on server item values. This mechanism allows the OPC server to notify a client when a server item value or quality has updated. This mechanism is called a *data change event*. Industrial Communication Toolbox supports data change event notification by executing a MATLAB function when a data change event is received from a connected OPC server. This section describes how to use the data change event notification.

Configure OPC Objects for Data Change Events

A data change event occurs at the `dagroup` object level. Using `dagroup` object properties, you can control whether a data change event is generated for a particular group, the minimum time between successive events, and the MATLAB function to run when the event notification is received and processed by Industrial Communication Toolbox. You can also control which items in a particular group should be monitored for data changes. In this way, you can control the number and frequency of data change events that MATLAB has to process. On a busy OPC server, you can also turn off data change notification for groups that you are not currently interested in.

The following sections describe how to control data change notification.

- "Control Data Change Notification for a Group" on page 7-8 describes how to turn off data change notification for a `dagroup` object.
- "Temporarily Disable Items in a Group" on page 7-9 describes how to control which items in a group must be monitored for data changes.
- "Customize the Data Change Event Response" on page 7-10 provided information on how to configure the MATLAB function to run when a data change event occurs.

Control Data Change Notification for a Group

The following properties of a `dagroup` object control whether a server notifies the group of data changes on items in that group:

- `UpdateRate`: The UpdateRate property defines the rate at which an OPC server must monitor server item values and generate data change events. Even if a server item's value changes more frequently than the update rate, the OPC server will only generate a data change at the interval specified by the update rate.
- `Subscription`: The Subscription property defines whether the OPC server will generate a data change event for the group. When you create a `dagroup` object, the `Subscription` property is set to `'on'`. When you set the `Subscription` property to `'off'`, you tell the OPC server not to generate data change events for that group.

- `Active`: The Active property must be `'on'` for data change events to be generated. When you create a `dagroup` object, the `Active` property is set to `'on'`. When you set the `Active` property to `'off'`, you remove any ability to read data from the group, whether through read operations or data change events.

A summary of group read, write, and data change behavior for the Active and Subscription properties is given in the following table.

Active	Subscription	Read	Write	Data Change
`'on'`	`'on'`	Yes	Yes	Yes
`'on'`	`'off'`	Yes	Yes	No
`'off'`	`'on'`	No	No	No
`'off'`	`'off'`	No	No	No

Temporarily Disable Items in a Group

You can temporarily disable items in a group without deleting the item from the group. When you disable a `daitem` object, the OPC server no longer monitors changes in the associated server item's value, and will therefore not generate data change events when the value of that server item changes.

You can disable a `daitem` object by setting that object's `Active` property to `'off'`. You can reenable the `daitem` object by setting the `Active` property to `'on'`.

Force a Data Change Event

You can force an OPC server to generate a data change event for all active items in a group by using the `refresh` function with the `dagroup` object as the first argument. The OPC server will generate a data change event containing information for every active item in the group.

You can pass an optional second argument to the `refresh` function to instruct the OPC server where to source the data values that are sent back in the data change event. By specifying a source of `'device'`, you instruct the OPC server to update the values from the device. By specifying a source of `'cache'` (the default) you instruct the OPC server to return values from the OPC server's cache.

How Data Change Events are Processed

Industrial Communication Toolbox software uses data change events for a number of tasks. The following activities take place when a data change event occurs:

1. The Value, Quality, and TimeStamp properties of the `daitem` object are automatically updated. For more information on these properties, see “OPC Data: Value, Quality, and TimeStamp” on page 8-2.
2. If the `dagroup` object is logging, the data change event is logged to memory and/or disk as a *record*. For information on logging, see “Log OPC Server Data” on page 7-11.
3. If the `dagroup` object's DataChangeFcn property is not empty, that function is called with the data change event information. By default, this property is empty, since data change events occur frequently. You can customize the behavior of the toolbox by setting this property to call a function that you choose. For information on the data change event, see the reference page for the DataChangeFcn property.

Note If you disable data change events by setting the Subscription property to `'off'` or the Active property to `'off'`, none of the activities listed above can take place. You cannot change the `Active` or `Subscription` properties while a `dagroup` object is logging, otherwise the logging task may never complete.

Customize the Data Change Event Response

One of the activities that occurs when Industrial Communication Toolbox software receives a data change event from the OPC server is the running of the function defined in the DataChangeFcn property. By setting this property to a the name of a function that you have written, you can fully customize the data change event behavior of the toolbox. For example, you may configure a `dagroup` object to monitor a server item that is updated from an operator interface. By pushing a button on the operator interface, the server item value will change, initiating a data change event on that group. By configuring the `DataChangeFcn` property to run a MATLAB function that performs control loop optimization, you can allow an operator to initiate a control loop performance test on all critical control loops in the plant.

Log OPC Server Data

In this section...

“How Data Is Logged” on page 7-11
“Configure a Logging Session” on page 7-13
“Execute a Logging Task” on page 7-15
“Get Logged Data into the MATLAB Workspace” on page 7-16

How Data Is Logged

The OPC Data Access Specification provides access to current values of data on an OPC server. Often, for analysis, troubleshooting, and prototyping purposes, you will want to know how OPC server data has changed over a period of time. For example, you can use time series data to perform control loop optimization or system identification on a portion of your plant. Industrial Communication Toolbox software provides a logging mechanism that stores a history of data that changed over a period of time. This section discusses how to configure and execute a logging task using the toolbox.

Note The toolbox software logging mechanism is not designed to replace a data historian or database application that logs data for an extended period. Rather, the logging mechanism allows you to quickly configure a task to log data on an occasional basis, where modifications to the plant-wide data historian may be unfeasible.

Industrial Communication Toolbox software uses the data change event to log data. Each data change event that is logged is called a *record*. The record contains information about the time the client logged the record, and details about each item in the data change event. Data change events are discussed in detail in “Data Change Events and Subscription” on page 7-8.

The use of a data change event for logging means that you should consider the following points when planning a logging session:

- **Logging takes place at the group level** — When planning a logging task, configure the group with only the items you need to log. Including more items than you need to will only increase memory and/or disk usage, and using that data may be more difficult due to unnecessary items in the data set.
- **Inactive items in a group will not be logged** — You must ensure that the items you need to log are active when you start a logging session. You control the active state of a `daitem` object using the `Active` property of the `daitem` object.
- **Data change events (records) may not include all items** — A data change event contains only the items in the group that have changed their value and/or quality state since the last update. Hence, a record is not guaranteed to contain every data item. You need to consider this when planning your logging session.
- **OPC logging tasks are not guaranteed to complete** — Because data change events only happen when an item in the group changes state on the server, it is possible to start a logging task that will never finish. For example, if the items in a group never change, a data change event will never be generated for that group. Hence, no records will be logged.
- **Logged data is not guaranteed to be regularly sampled** — It is possible to force a data change event at any time (see “Force a Data Change Event” on page 7-9). If you do this during a

logging task, the data change events may occur at irregular sample times. Also, a data change event may not contain information for every item in the group. Consequently, logged OPC server data may not occur at regular sample times.

An overview of the logging task, and a representation of how the above points impact the logging session, is provided in the following section.

Overview of a Logging Task

To illustrate a typical logging task, the following example logs to disk and memory six records of data from two items provided by the Matrikon OPC Simulation Server. During the logging task, data is retrieved from memory. When the task stops, the remaining records are retrieved.

Step 1: Create the OPC object hierarchy

This example creates a hierarchy of OPC objects for two items provided by the Matrikon Simulation Server. To run this example on your system, you must have the Matrikon Simulation Server installed. Alternatively, you can replace the values used in the creation of the objects with values for a server you can access.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da,'CallbackTest');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-Toothed Waves.Boolean');
```

Step 2: Configure the logging duration

This example sets the `UpdateRate` value to 1 second, and the `RecordsToAcquire` property to 6. See “Control the Duration of a Logging Session” on page 7-13 for more information on this step.

```
grp.UpdateRate = 1;
grp.RecordsToAcquire = 6;
```

Step 3: Configure the logging destination

In this example, data is logged to disk and memory. The disk filename is set to `LoggingExample.olf`. The LogToDiskMode property is set to `'overwrite'`, so that if the filename exists, the toolbox engine must overwrite the file. See “Control the Logged Data Destination” on page 7-14 for more information on this step.

```
grp.LoggingMode = 'disk&memory';
grp.LogFileName = 'LoggingExample.olf';
grp.LogToDiskMode = 'overwrite';
```

Step 4: Start the logging task

Start the `dagroup` object. The logging task is started, and the group summary updates to reflect the logging status. See “Start a Logging Task” on page 7-15 for more information on this step.

```
start(grp)
grp
```

Step 5: Monitor the Logging Progress

After about 3 seconds, retrieve and show the last acquired value. After another second, obtain the first two records during the logging task. Then wait for the logging task to complete. See “Monitor the Progress of a Logging Task” on page 7-15 for more information on this step.

```
pause(3.5)
sPeek = peekdata(grp, 1);
% Display the local event time, item IDs and values
disp(sPeek.LocalEventTime)
disp({sPeek.Items.ItemID;sPeek.Items.Value})
pause(1)
sGet = getdata(grp, 2);
wait(grp)
```

Step 6: Retrieve the data

This example retrieves the balance of the records into a structure array. See “Retrieve Data from Memory” on page 7-17 for more information on this step.

```
sFinished = getdata(grp,grp.RecordsAvailable);
```

Step 7: Clean up

When you no longer need them, always remove from memory any toolbox objects and the variables that reference them. Deleting the `opcda` client object also deletes the group and `daitem` objects.

```
disconnect(da)
delete(da)
clear da grp itm1 itm2
```

Configure a Logging Session

A logging session is associated with a `dagroup` object. Before you start a logging session, you will need to ensure that the logging session is correctly configured. This section explains how you can control

- The duration of a logging session (see “Control the Duration of a Logging Session” on page 7-13). By default, a group will log approximately one minute of data at half second intervals.
- The destination of logged data (see “Control the Logged Data Destination” on page 7-14). By default, a group will log data to memory.
- The response to events that take place during a logging session (see “Configure Logging Callbacks” on page 7-15). By default, a logging session takes no action in response to events that take place during a logging session.

Control the Duration of a Logging Session

While you cannot guarantee that a logging session will take a specific amount of time (see “How Data Is Logged” on page 7-11), you can control the rate at which the server will update the items and how many records the logging task should store before automatically stopping the logging task. You control these aspects of a logging task by using the following properties of the `dagroup` object:

- `UpdateRate`: The UpdateRate property defines how often the item values are inspected.
- `RecordsToAcquire`: The RecordsToAcquire property defines how many records the toolbox must log before automatically stopping a logging session. A logging task can also be stopped manually, using the `stop` function.
- `DeadbandPercent`: The DeadbandPercent property does not control the duration of a logging task directly, but has a significant influence over how often a data change event is generated for *analog items* (an item whose value is not confined to discrete values). By setting the `DeadbandPercent` property to 0, you can ensure that a data change event occurs each time a value changes. For more information on DeadbandPercent, consult the property reference page.

You can use the `UpdateRate` and `RecordsToAcquire` properties to define the minimum duration of a logging task. The duration of a logging task is at least

`UpdateRate` * `RecordsToAcquire`

For example, if the `UpdateRate` property is 10 (seconds) and the `RecordsToAcquire` property is 360, then provided that a data change event is generated each time the server queries the item values, the logging task will take 3600 seconds, or one hour, to complete.

Control the Logged Data Destination

Industrial Communication Toolbox software allows you to log data to memory, to a disk file, or both memory and a disk file. When logging data to memory, you can log only as much data as will fit into available memory. Also, if you delete the `dagroup` object that logged the data without extracting that data to the MATLAB workspace, the data will be lost. The advantage of logging data to memory is that logging to memory is faster than using a disk file.

Logging data to a disk file usually means that you can log more data, and the data is not lost if you quit MATLAB or delete the `dagroup` object that logged the data. However, reading data from a disk file is slower than reading data from memory.

The LoggingMode property of a `dagroup` object controls where logged data is stored. You can specify `'memory'` (the default value), or `'disk'`, or `'disk&memory'` as the value for `LoggingMode`.

The following properties control how the toolbox logs data to disk. You must set the `LoggingMode` property to `'disk'` or `'disk&memory'` for these properties to take effect:

- `LogFileName`: The LogFileName property is a character vector that specifies the name of the disk file that is used to store logged data. If the file does not exist, data will be logged to that filename. If the file does exist, the `LogToDiskMode` property defines how the toolbox behaves.
- `LogToDiskMode`: The LogToDiskMode property controls how the toolbox handles disk logging when the file specified by `LogFileName` already exists. Each time a logging task is started, if the `LoggingMode` is set to `'disk'` or `'disk&memory'`, the toolbox checks to see if a file with the name specified by the `LogFileName` property exists. If the file exists, the toolbox will take the following action, based on the `LogToDiskMode` property:

  - `'append'`: When `LogToDiskMode` is set to `'append'`, logged data will be added to the existing data in the file.
  - `'overwrite'`: When `LogToDiskMode` is set to `'overwrite'`, all existing data in the file will be removed without warning, and new data will be logged to the file.
  - `'index'`: When `LogToDiskMode` is set to `'index'`, the toolbox automatically changes the log filename, according to the following algorithm:

    The first log filename attempted is specified by the initial value of `LogFileName`.

    If the attempted filename exists, `LogFileName` is modified by adding a numeric identifier. For example, if `LogFileName` is initially specified as `'groupRlog.olf'`, then `groupRlog.olf` is the first attempted filename, `groupRlog01.olf` is the second filename, and so on. If `LogFileName` already contains numeric characters, they are used to determine the next sequence in the modifier. For example, if the `LogFileName` is initially specified as `'groupRlog010.olf'`, and `groupRlog010.olf` exists, the next attempted file is `groupRlog011.olf`, and so on.

The actual filename used is the first filename that does not exist. In this way, each consecutive logging operation is written to a different file, and no previous data is lost.

Configure Logging Callbacks

You can configure the `dagroup` object so that MATLAB will automatically execute a function when the logging task starts, when the logging task stops, and each time a specified number of records is acquired during a logging task. The `dagroup` object has three *callback properties* that are used during a logging session. Each callback property defines the action to take when a particular logging event occurs:

- **Start event**: A start event is generated when a logging task starts.
- **Records acquired event**: A records acquired event is generated each time a logging task acquires a set number of records.
- **Stop event**: A stop event is generated when a logging task stops, either automatically, or by the user calling the `stop` function.

For an example of using callbacks in a logging task, see “View Recently Logged Data” on page 9-15.

Execute a Logging Task

Once you have configured your logging task you can execute the task. Executing a logging task involves starting the logging task, monitoring the task progress, and stopping the logging task.

Start a Logging Task

You start a logging task by calling the `start` function, passing the `dagroup` object that you want to start logging. The following example starts a logging task for the `dagroup` object `grp`.

```
start(grp)
```

When you start a logging task, certain group and item properties become read-only, as modifying these properties during a logging task would corrupt the logging process. Also, the `dagroup` object performs the following operations:

1. Generates a start event and executes the `StartFcn` callback.
2. If `Subscription` is `'off'`, sets `Subscription` to `'on'` and issues a warning.
3. Removes all records associated with the object from the toolbox engine.
4. Sets `RecordsAcquired` and `RecordsAvailable` to `0`.
5. Sets the `Logging` property to `'on'`.

Monitor the Progress of a Logging Task

During a logging task, you can monitor the progress of the task by examining the following properties of the `dagroup` object:

- `Logging`: The Logging property is set to `'on'` at the start of a logging task, and set to `'off'` when the logging task stops.
- `RecordsAcquired`: The RecordsAcquired property contains the number of records that have been logged to the destination specified by the `LoggingMode` property. When a start function is called, `RecordsAcquired` is set to `0`. When `RecordsAcquired` reaches `RecordsToAcquire`, the logging task stops automatically.

- `RecordsAvailable`: The RecordsAvailable property contains the number of records that have been stored in the toolbox engine for this logging task. Data is only logged to memory if the `LoggingMode` is set to `'memory'` or `'disk&memory'`. You extract data from the toolbox engine using the `getdata` function. See "Get Logged Data into the MATLAB Workspace" on page 7-16 for more information on using `getdata`.

You can monitor these properties in the summary display of a `dagroup` object, by typing the name of the `dagroup` object at the command line.

```
grp
```

```
grp =
Summary of OPC Data Access Group Object: group1
   Object Parameters
      Group Type   : private
      Item         : 1-by-1 daitem object
      Parent       : localhost/Matrikon.OPC.Simulation.1
      Update Rate  : 0.5
      Deadband     : 0%
   Object Status
      Active       : on
      Subscription : on
      Logging      : on
   Logging Parameters
      Records      : 120
      Duration     : at least 60 seconds
      Logging to   : disk
      Log File     : group1log.olf ('index' mode)
      Status       : 5 records acquired since starting.
                     0 records available for GETDATA/PEEKDATA
```

Stop a Logging Task

A logging task stops when one of the following conditions is met:

- The number of records logged reaches the value defined by the `RecordsToAcquire` property.
- You manually stop the logging task by using the `stop` function.

The following example manually stops the logging task for `dagroup` object `grp`.

```
stop(grp)
```

When a logging task stops, the `Logging` property is set to `'off'`, a stop event is generated, and the `StopFcn` callback is executed.

Get Logged Data into the MATLAB Workspace

Industrial Communication Toolbox software does not log data directly to the MATLAB workspace. When logging to memory, the data is buffered in the toolbox engine in a storage-efficient way. When logging to disk, the data is logged in ASCII format. To analyze your data, you need to extract the data from the toolbox engine or from a disk file into MATLAB for processing. This section describes how to get your logged data into the MATLAB workspace. The following sections describe this process:

- "Retrieve Data from Memory" on page 7-17, discusses how to retrieve data from the toolbox engine into MATLAB.

- “Retrieve Data from Disk” on page 7-18, discusses how to retrieve data from a disk file into MATLAB.

Whether you log data to memory or to disk, you can retrieve that logged data in one of two formats:

- Structure format: This format stores each data change event in a structure. Data from a logging task is simply an array of such structures.
- Array format: To visualize and analyze your data, you will need to work with the time series of each of the items in the group. The array format is the logged structure data, “unpacked” into separate arrays for the Value, Quality, and TimeStamp.

Retrieve Data from Memory

You retrieve data from memory using the `getdata` function, passing the `dagroup` object as the first argument, and the number of records you want to retrieve as the second argument. The data is returned as a structure containing data from each data change event in the logging task. For example, to retrieve 20 records for the `dagroup` object `grp`:

```
s = getdata(grp, 20);
```

If you do not supply a second argument, `getdata` will try to retrieve the number of records specified by the RecordsToAcquire property of the `dagroup` object. If the toolbox engine contains fewer records for the group than the number requested, a warning is generated and all of the available records will be retrieved.

To retrieve data in array format, you must indicate the data type of the returned values. You pass a character vector defining that data type as an additional argument to the `getdata` function. Valid data types are any MATLAB numeric data type (for example, `'double'` or `'uint32'`) plus `'cell'` to denote the MATLAB cell array data type.

When you specify a numeric data type or cell array as the data type for `getdata`, the logged data is returned in separate arrays for the item IDs logged, the value, quality, time stamp, and the local event time of each data change event logged. You must therefore specify up to five output arguments for the `getdata` function when retrieving data in array format.

For example, to retrieve 20 records of logged data in double array format from `dagroup` object `grp`.

```
[itmID,val,qual,tStamp,evtTime] = getdata(grp,20,'double');
```

Once you have retrieved data to the MATLAB workspace using `getdata`, the records are removed from the toolbox engine to free up memory for additional logged records. If you specify a smaller number of records than those available in memory, `getdata` will retrieve the oldest records. You can use the RecordsAvailable property of the `dagroup` object to determine how many records the toolbox engine has stored for that group.

During a logging task, you can examine the most recently acquired records using the `peekdata` function, passing the `dagroup` object as the first argument, and the number of records to retrieve as the second argument. Data is returned in a structure. You cannot return data into separate arrays using `peekdata`. You can convert the structure returned by `peekdata` into separate arrays using the `opcstruct2array` function. Data retrieved using `peekdata` is not removed from the toolbox engine.

For an example of using `getdata` and `peekdata` during a logging task, see “Overview of a Logging Task” on page 7-12.

When you delete a `dagroup` object, the data stored in the toolbox engine for that object is also deleted.

Retrieve Data from Disk

You can retrieve data from a disk file into the MATLAB workspace using the `opcread` function. You pass the name of the file containing the logged OPC data as the first argument. The data stored in the log file is returned as a structure array, in the same format as the structure returned by `getdata`. Records retrieved from a log file into the MATLAB workspace are not removed from the log file.

You can specify a number of additional arguments to the `opcread` function, that control the records that are retrieved from the file. The additional arguments must be specified by an option name and the option value. The following options are available.

Option Name	Option Value Description
`'items'`	Specify a cell array of item IDs that you want returned. Items not in this list will not be read.
`'dates'`	Specify a date range for the event times. The range must be `[startDt endDt]` where `startDt` and `endDt` are MATLAB date numbers.
`'records'`	Specify the index of records to retrieve as `[startRec endRec]`. Records outside these indices will not be read.
`'datatype'`	Specify the data type, as a character vector, that should be used for the returned values. Valid data type character vectors are the same as for `getdata`. If you specify a numeric data type or `'cell'`, the output will be returned in separate arrays. If you specify a numeric array data type such as `'double'` or `'uint32'`, and the logged data contains arrays or character vectors, an error will be generated and no data will be returned.

The following example retrieves the data logged during the example on page “Overview of a Logging Task” on page 7-12, first into a structure array, and then records 3 to 6 are retrieved into separate arrays for Value, Quality, and TimeStamp.

```
sDisk = opcread('LoggingExample.olf')
```

```
sDisk =
40x1 struct array with fields:
    LocalEventTime
    Items

[i,v,q,t,e] = opcread('LoggingExample.olf', ...
    'records',[3,6], 'datatype','double')
i =
    'Random.Real8'    'Random.UInt2'    'Random.Real4'
v =
  1.0e+004 *
    0.7819    3.0712    1.4771
    1.5599    2.7792    2.2051
    1.4682    0.4055    0.5315
    0.0235    2.4473    1.5456
q =
  'Good: Non-specific'  'Good: Non-specific'  'Good: Non-specific'
  'Good: Non-specific'  'Good: Non-specific'  'Good: Non-specific'
  'Good: Non-specific'  'Good: Non-specific'  'Good: Non-specific'
  'Good: Non-specific'  'Good: Non-specific'  'Good: Non-specific'
```

```
t =
  1.0e+005 *
    7.3202    7.3202    7.3202
    7.3202    7.3202    7.3202
    7.3202    7.3202    7.3202
    7.3202    7.3202    7.3202
e =
  1.0e+005 *
    7.3202
    7.3202
    7.3202
    7.3202
```

Note For a record to be returned by `opcread`, it must satisfy all the options passed to `opcread`.

Working with OPC Data

When an OPC server returns data from a read or logging operation, three pieces of information make up the data. The Value, Quality, and Timestamp all contribute information about the data point that is returned. As a result, you need to understand how to deal with this information together, because one aspect of the data in isolation will not provide a complete picture of the data returned by a read operation, data change event, read async event, or toolbox logging task.

This chapter describes how Industrial Communication Toolbox software handles data returned by an OPC server.

- “OPC Data: Value, Quality, and TimeStamp” on page 8-2
- “Work with Structure-Formatted Data” on page 8-6
- “Array-Formatted Data” on page 8-11
- “Work with Different Data Types” on page 8-13

OPC Data: Value, Quality, and TimeStamp

In this section...

"Introduction to OPC Data" on page 8-2
"Relationship Between Value, Quality, and TimeStamp" on page 8-2
"How Value, Quality, and TimeStamp Are Obtained" on page 8-3

Introduction to OPC Data

OPC servers provide access to many server items. To reduce network traffic between the server and the "device" associated with each server item (a field instrument, or a memory location in a PLC, SCADA, or DCS system) the OPC server stores information about each server item in the server's "cache," updating that information only as frequently as required to satisfy the requests of all clients connected to that server. Because this process results in data in the cache that may not reflect the actual value of the device, the OPC server provides the client with additional information about that value.

This section describes the OPC Value, Quality, and TimeStamp properties, and how they should be used together to assess the information provided by an OPC server.

Relationship Between Value, Quality, and TimeStamp

Every server item on an OPC server has three properties that describe the status of the device or memory location associated with that server item:

- **Value** — The `Value` of the server item is the last value that the OPC server stored for that particular item. The value in the cache is updated whenever the server reads from the device. The server reads values from the device at the update rate specified by the `dagroup` object's `UpdateRate` property, and only when the item and group are both active. You control the active status of an item or group using that object's `Active` property.

  In addition, for analog type data (data with the additional OPC Foundation Recommended Properties `'High EU'` and `'Low EU'`) the percentage change between the cached value and the device value must exceed the `DeadbandPercent` property specified for that item in order for the cached value to be updated.
- **Quality** — The `Quality` of the server item is a character vector that represents information about how well the cache value matches the device value. The Quality is made up of two parts: a major quality, which can be `'Good'`, `'Bad'`, or `'Uncertain'`, and a minor quality, which describes the reason for the major quality. For more information on `Quality`, see "OPC Quality" on page A-2.

  The `Quality` of the server item can change without the `Value` changing. For instance, if the OPC server attempts to obtain a `Value` from the device but that operation fails, the `Quality` will be set to `'Bad'`. Also, when you change the client's Active property, the `Quality` will change.

  You must always examine the `Quality` of an item before using the `Value` property of that item.
- **TimeStamp** — The `TimeStamp` of a server item represents the most recent time that the server assessed that the device set the `Value` and `Quality` properties of that server item. The `TimeStamp` can change without the `Value` changing. For example, if the OPC server obtains a value from the device that is the same as the current `Value`, the `TimeStamp` property will still be updated, even if the `Value` property is not.

Industrial Communication Toolbox software provides access to the `Value`, `Quality`, and `TimeStamp` properties of a server item through properties of the OPC `daitem` object associated with that server item.

How Value, Quality, and TimeStamp Are Obtained

Industrial Communication Toolbox provides all three OPC Data Access Standard mechanisms for reading data from an OPC server. The toolbox uses these three mechanisms in various ways to return data from those functions, to provide event information, to update properties of toolbox objects, and to log data to memory and disk.

The toolbox uses the three OPC Data Access mechanisms as described in the following sections:

- "OPC Data Returned from Synchronous Read Operations" on page 8-3 describes the synchronous read mechanism used by the `read` function.
- "OPC Data Returned in Asynchronous Read Operations" on page 8-3 describes the asynchronous read mechanism used by the `readasync` function.
- "OPC Data Returned from a Data Change Event" on page 8-4 describes the data change event notification mechanism used with subscribed, active groups, with the refresh function, and by the toolbox logging process.

OPC Data Returned from Synchronous Read Operations

You initiate a synchronous read operation by using the `read` function. When you read from a `dagroup` object, all items in that group are read in one instruction.

You can specify the source of a synchronous read operation as `'cache'` or `'device'`. If you read from the cache, the server simply returns the value in the cache. If you read from the device, the server will get the value from the device and update the cache before sending the Value, Quality, and TimeStamp information back as part of the read operation.

Industrial Communication Toolbox returns the data in the output structure from the read operation. Each element of the structure array contains information about one of the items read.

Whenever you read values using the read function, the toolbox updates the `daitem` object's `Value`, `Quality`, and `TimeStamp` properties with the values read from the server.

OPC Data Returned in Asynchronous Read Operations

You initiate an asynchronous read operation by using the `readasync` function. When you read from a `dagroup` object, all items in that group are read in one instruction.

Asynchronous read operations always use the device as the source of the read. Whenever you send an asynchronous read request, the server will read values from the devices connected to the items. The server will then update that server item's Value, Quality, and TimeStamp in the cache before sending an asynchronous read event back to the toolbox.

The toolbox returns information from an asynchronous read operation via the read async event structure. This event structure is stored in the `opcda` client object's event log, which you can access using the `EventLog` property of the client. The event structure is also passed to the callback function defined in the `ReadAsyncFcn` property of the `dagroup` object that initiated the asynchronous read operation. For more information on the format of the event structures, see "Event Structures" on page 9-8.

When an asynchronous read operation succeeds, in addition to returning data via the event structures, the toolbox also updates the `Value`, `Quality`, and `TimeStamp` properties of the associated `daitem` object.

OPC Data Returned from a Data Change Event

The third mechanism for getting data from an OPC server involves the data change event. The OPC server generates a data change event for a group at the period specified by the UpdateRate property when the Value or Quality of an item in the group changes. You do not have to specifically request a data change event, because the OPC server will automatically generate a data change event. However, you can force a data change event at any time using the `refresh` function.

An OPC server will generate a data change event only for an active, subscribed group containing active items. You control the active status of `dagroup` objects and `daitem` objects by setting their `Active` property. You control the subscribed status of a `dagroup` object by setting the `Subscription` property of the `dagroup` object.

The following points describe how an OPC server generates a data change event:

- When you configure a group, you define the rate at which the server must scan items in that group. This rate is controlled by the `UpdateRate` property for a `dagroup` object. The server updates the Value, Quality, and TimeStamp values in the cache for the items in that group at the required update rate. Note that if a device cannot provide a value in that time, the server may reduce the rate at which it updates the value in the server cache for that item.
- If you set an item's `Active` property to `'off'`, the server will stop scanning that item. You must set the `Active` property to `'on'` for the server to scan the item again.
- If you set the Active property of a `dagroup` object to `'off'`, the server will stop scanning all items in that group. You can still perform asynchronous read operations, and synchronous read operations from the `'device'`, but no operations involving the server cache can be performed. You must set the `Active` property to `'on'` to enable operations involving the server cache.
- If the `Subscription` property for a `dagroup` object is set to `'on'`, then every time the server updates cache values for the items in that group, the server will send a data change event for that group, to the client object. The data change event contains information about every item whose Value, Quality, or TimeStamp updated.
- If you set the `Subscription` property to `'off'`, then the OPC server will not generate data change events. However, as long as the group is still active, the OPC server will continue to scan all active items for that group, at the rate specified by the `UpdateRate` property.

When the OPC server generates a data change event, the toolbox performs the following tasks:

1. The `daitem` object `Value`, `Quality`, and `TimeStamp` properties are updated for each item that is included in the data change event.
2. The callback function defined by the `DataChangeFcn` property of the `dagroup` object is called. For more information on callbacks, see “Create and Execute Callback Functions” on page 9-12.
3. If the group is logging data, the data change event is stored in memory and/or on disk. For more information on logging, see “Log OPC Server Data” on page 7-11.
4. If the group is logging, and the number of records acquired is a multiple of the `RecordsAcquiredFcnCount` property of the `dagroup` object, then the callback function defined by the `RecordsAcquiredFcn` property of the `dagroup` object is called. For more information on callbacks, see “Create and Execute Callback Functions” on page 9-12.

For more information on the structure of a data change event, see “Data Fields for Cancel Async, Data Change, Error, Read Async, and Write Async Events” on page 9-8.

Work with Structure-Formatted Data

In this section...

"When Structures Are Used" on page 8-6
"Perform a Read Operation on Multiple Items" on page 8-6
"Interpret Structure-Formatted Data" on page 8-7
"When to Use Structure-Formatted Data" on page 8-9
"Convert Structure-Formatted Data to Array Format" on page 8-9

When Structures Are Used

Industrial Communication Toolbox software uses structures to return data from an OPC server, for the following operations:

- Synchronous read operations, executed using the `read` function.
- Asynchronous read operations, executed using the `readasync` function.
- Data change events generated by the OPC server for all active, subscribed groups or through a `refresh` function call.
- Retrieving logged data in structure format from memory using the `getdata` or `peekdata` functions.

In all cases, the structure of the returned data is the same. This section describes that structure, and how you can use the structure data to understand OPC operations.

Perform a Read Operation on Multiple Items

To illustrate how to use structure-formatted data, the following example reads values from three items on the Matrikon OPC Simulation Server.

Step 1: Create OPC Group Objects

This example creates a hierarchy of OPC objects for the Matrikon Simulation Server. To run this example on your system, you must have the Matrikon Simulation Server installed. Alternatively, you can replace the values used in the creation of the objects with values for a server you can access.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da,'StructExample');
itm1 = additem(grp,'Random.Real8');
itm2 = additem(grp,'Saw-toothed Waves.UInt2');
itm3 = additem(grp,'Random.Boolean');
```

Step 2: Read Data

This example reads values first from the device and then from the server cache. The data is returned in structure format.

```
r1 = read(grp, 'device');
r2 = read(grp);
```

Step 3: Interpret the Data

The data is returned in structure format. To interpret the data, you must extract the relevant information from the structures. In this example, you compare the Value, Quality, and TimeStamp to confirm that they are the same for both read operations.

```
disp({r1.ItemID;r1.Value;r2.Value})
disp({r1.ItemID;r1.Quality;r2.Quality})
disp({r1.ItemID;r1.TimeStamp;r2.TimeStamp})
```

Step 4: Read More Data

By reading first from the cache and then from the device, you can compare the returned data to see if any change has occurred. In this case, the data will not be the same.

```
r3 = read(grp);
r4 = read(grp, `device');
disp({r3.ItemID;r3.Value;r4.Value})
```

Step 5: Clean Up

Always remove toolbox objects from memory, and the variables that reference them, when you no longer need them.

```
disconnect(da)
delete(da)
clear da grp itm1 itm2 itm3
```

Interpret Structure-Formatted Data

All data returned by the `read`, `opcread`, and `getdata` functions, and included in the data change and read async event structures passed to callback functions, has the same underlying format. The format is best explained by starting with the output from the `read` function, which provides the basic building block of structure-formatted data.

Structure-Formatted Data for a Single Item

When you execute the `read` function with a single `daitem` object, the following structure is returned.

```
rSingle = read(itm1)

rSingle =

       ItemID: 'Random.Real8'
        Value: 1.0440e+004
      Quality: 'Good: Non-specific'
    TimeStamp: [2004 3 10 14 46 9.5310]
        Error: ''
```

All structure-formatted data for an item will contain the `ItemID`, `Value`, `Quality`, and `TimeStamp` fields.

Note The `Error` field in this example is specific to the `read` function, and is used to indicate any error message the server generated for that item.

Structure-Formatted Data for Multiple Items

If you execute the `read` function with a group object containing more than one item, a structure array is returned.

```
rGroup = read(grp)

rGroup = 

3x1 struct array with fields:
    ItemID
    Value
    Quality
    TimeStamp
    Error
```

In this case, the structure array contains one element for each item that was read. The `ItemID` field in each element identifies the item associated with that element of the structure array.

Note When you perform asynchronous read operations, and for data change events, the order of the items in the structure array is determined by the OPC server. The order may not be the same as the order of the items passed to the read function.

Structure-Formatted Data for Events

Event structures contain information specifically about the event, as well as the data associated with that event.

The following example displays the contents of a read async event.

```
cleareventlog(da);
tid = readasync(itm);
% Wait for the read async event to occur
pause(1);
event = get(da, 'EventLog')

event = 

    Type: 'ReadAsync'
    Data: [1x1 struct]
```

The Data field of the event structure contains

```
event.Data

ans = 

    LocalEventTime: [2004 3 11 10 59 57.6710]
           TransID: 4
         GroupName: 'StructExample'
             Items: [1x1 struct]
```

The Items field of the Data structure contains

```
event.Data.Items
```

```
ans = 

       ItemID: 'Random.Real8'
        Value: 9.7471e+003
      Quality: 'Good: Non-specific'
    TimeStamp: [2004 3 11 10 59 57.6710]
```

From the example, you can see that the event structure embeds the structure-formatted data in the `Items` field of the `Data` structure associated with the event. Additional fields of the `Data` structure provide information on the event, such as the source of the event, the time the event was received by the toolbox, and the transaction ID of that event.

Structure-Formatted Data for a Logging Task

Industrial Communication Toolbox software logs data to memory and/or disk using the data change event. When you return structure-formatted data for a logging task using the `opcread` or `getdata` function, the returned structure array contains the data change event information arranged in a structure array. Each element of the structure array contains a *record*, or data change event. The structure array has the `LocalEventTime` and `Items` fields from the data change event. The `Items` field is in turn a structure array containing the fields `ItemID`, `Value`, `Quality`, and `TimeStamp`.

When to Use Structure-Formatted Data

For the read, read async and data change events, you must use structure-formatted data. However, for a logging task, you have the option of retrieving the data in structure format, or numeric or cell array format.

For a logging task, you should use structure-formatted data when you are interested in

- The “raw” event information returned by the OPC server. The raw information may help in diagnosing the OPC server configuration or the client configuration. For example, if you see a data value that does not change frequently, yet you know that the device should be changing frequently, you can examine the structure-formatted data to determine when the OPC server notifies clients of a change in Value, Quality and/or TimeStamp.
- Timing information rather than time series data. If you need to track when an operator changed the state of a switch, structure-formatted data provides you with event-based data rather than time series data.

For other tasks that involve time series data, such as visualization of the data, analysis, modeling, and optimization operations, you should consider using the cell or numeric array output format for `getdata` and `opcread`. For more information on array formats, see “Array-Formatted Data” on page 8-11.

Convert Structure-Formatted Data to Array Format

If you retrieve data from memory or disk in structure format, you can convert the resulting structure into array format using the `opcstruct2array` function. You pass the structure array to the function, and it will return the `ItemID`, `Value`, `Quality`, `TimeStamp`, and `EventTime` information contained in that structure array.

The `opcstruct2array` function is particularly useful when you want to visualize or analyze time series data without removing it from memory. Because `peekdata` only returns structure arrays (due to speed considerations), you can use `opcstruct2array` to convert the contents of the structure data into separate arrays for visualization and analysis purposes.

Note You should always retrieve data in numeric or cell array format whenever you only want to manipulate the time series data. Although the `opcstruct2array` function has been designed to use as little memory as possible, conversion in MATLAB software still requires storage space for both the structure array and the resulting arrays.

For an example of using `opcstruct2array`, see "Write a Callback Function" on page 9-12.

Array-Formatted Data

In this section...
“Array Content” on page 8-11
“Conversion of Logged Data to Arrays” on page 8-11

Array Content

Industrial Communication Toolbox software can return arrays of Value, Quality, and TimeStamp information from a logging task. You can retrieve arrays from memory using `getdata`, or from disk using `opcread`, by specifying the data type as `'cell'` or any MATLAB numeric array data type, such as `'double'` or `'uint32'`. Consult the function reference pages for details on how to specify the data type.

When you request array-formatted data, the toolbox returns arrays of each of the following elements of the records in memory or on disk:

- `ItemID` — A 1-by-*`nItems`* list of all item IDs occurring in the structure array. Each record is searched and all unique item IDs are returned in a cell array. The order of the item IDs must be used to interpret any of the Value, Quality, or TimeStamp arrays.
- `Value` — An *`nRecs`*-by-*`nItems`* array of values for each item ID defined in the `ItemID` variable, at each time stamp defined by the `TimeStamp` array. Each column of the `Value` array represents the history of values for the corresponding item in the `ItemID` array. Each row corresponds to one record. See “Treatment of Missing Data” on page 8-12 for information on how the `Value` array is populated.
- `Quality` — An *`nRecs`*-by-*`nItems`* cell array of character vectors. Each column represents the history of qualities for the corresponding item in the `ItemID` array. Each row corresponds to the qualities for a particular record. If a particular item ID was not part of a record (because the item did not change during that period), the corresponding column in that row is set to `'Repeat'`.
- `TimeStamp` — An *`nRecs`*-by-*`nItems`* array of time stamps for each value in the `Value` field. The time stamps are in MATLAB date number format. For more information on MATLAB date numbers, see the `datenum` function help.
- `EventTime` — An *`nRecs`*-by-1 array of times that the record was received by the toolbox (the `LocalEventTime` field of the record in structure format). The times are in MATLAB date number format. For more information on MATLAB date numbers, see the `datenum` function help.

Conversion of Logged Data to Arrays

When you request array-formatted data from `getdata` or `opcread`, you must define the desired data type for the returned `Value` array. Industrial Communication Toolbox automatically converts each record of logged data from the item's data type (defined by the `DataType` property of that item) to the requested data type.

When converting logged data to arrays, the toolbox must consider two factors when populating the returned arrays:

- A record may not contain information for every item in the logging task. “Treatment of Missing Data” on page 8-12 discusses how the toolbox deals with missing data.

- A record may contain an array value for a single item. Such values cannot easily be converted to a single value of numeric data types. “Treatment of Array Data Values” on page 8-12 discusses how the toolbox deals with this issue.

Treatment of Missing Data

When the toolbox logs data, each logged record may not contain all items in the logging task. When converting the data to array format, every item involved in the logging task must be allocated a value, a quality, and a time stamp for each record. Therefore, in a logging task there may be "missing" data for a particular item in a particular record. The toolbox uses the following rules to determine how to fill the missing entry in each array:

- `Value` — When you request the `'cell'` array data type, the value used for the missing entry is an empty double array (`[]`). When requesting a numeric data type, the value used for the missing entry is the last value for that item. If no previous value is known, the equivalent `NaN` (not a number) entry is used. For example, if the very first record does not contain an entry for that item, `NaN` is used to fill in the missing entry in the first row of the `Value` array. The equivalent `NaN` value for integer and logical data types is `0`.
- `Quality` — The missing entry is filled with the specific quality of `'Repeat'`.
- `TimeStamp` — The time stamp used for the missing entry is the first time stamp found in that particular record (row).

Treatment of Array Data Values

For each record stored in memory or on disk during a logging task, a single item may return an array of values. When converting logged data to array format, each item in each record has only one entry in the `Value` array allocated to that record and item.

For the `'cell'` data type, the toolbox is able to store the array returned as the Value for that element, because a MATLAB cell array is able to store any data type of any size in each element of the cell array.

For numeric data types, such as `'double'` or `'uint32'`, the resulting `Value` array provides space for only a single value. Consequently, if an array value is found in a logging task, and you have requested a numeric array data type, an error will be generated. You must use the `'cell'` data type or the structure format to return logged data that contains arrays as values.

Work with Different Data Types

In this section...

"Conversion Between MATLAB Data Types and COM Variant Data Types" on page 8-13
"Conversion of Values Written to an OPC Server" on page 8-14
"Conversion of Values Read from an OPC Server" on page 8-14
"Handling Arrays for Item Values" on page 8-15

Conversion Between MATLAB Data Types and COM Variant Data Types

The OPC Data Access Standard uses the Microsoft COM Specification for communication between the OPC server and OPC client. A significant amount of the data exchanged between the OPC server and the client is the value from a server item or the value that a client wants to write to a server item. The Microsoft COM Specification uses Microsoft Variants to send different data types between the client and server. This section discusses how Industrial Communication Toolbox software converts MATLAB data types to COM Variants when writing values, and COM Variants to MATLAB data types when reading values.

OPC servers require all values to be written to server items in COM Variant format. The server also provides the toolbox with COM Variants when an item's `Value` property is read or returned by the server. The toolbox automatically converts between the COM Variant type and MATLAB data types according to the table shown below.

Table 8-1, Conversion from MATLAB Data Type to COM Variant Data Type

MATLAB Data Type	OPC Server Data Type (COM Variant Type)	Remarks
`double`	`VT_R8`	
`single`	`VT_R4`	
`char`	`VT_BSTR`	
`logical`	`VT_BOOL`	
`uint8`	`VT_UI1`	
`uint16`	`VT_UI2`	
`uint32`	`VT_UI4`	
`uint64`	`VT_UI8`	
`int8`	`VT_I1`	
`int16`	`VT_I2`	
`int32`	`VT_I4`	
`int64`	`VT_I8`	
`function_handle`	`N/A`	Not allowed
`cell`	N/A	Not allowed
`struct`	N/A	Not allowed
`object`	N/A	Not allowed
N/A	`VT_DISPATCH`	Not allowed
N/A	`VT_BYREF`	Not allowed
`double`	`VT_EMPTY`	Returns the empty matrix (`[]`)

Conversion of Values Written to an OPC Server

When you write values to the OPC server using the `write` or `writeasync` function, you can provide any MATLAB data for the write operation. When you write data to an OPC server, the following data conversions take place:

1. Industrial Communication Toolbox software converts the value into the equivalent COM Variant according to Table 8-1, Conversion from MATLAB Data Type to COM Variant Data Type. If any disallowed data type is encountered (for example, if you attempt to write a MATLAB structure), an error will be generated.
2. The COM Variant is sent to the OPC server.
3. The OPC server will attempt to convert the COM Variant to the server item's canonical data type, using COM Variant conversion rules. If the conversion fails, the server will return an error.

Conversion of Values Read from an OPC Server

When an OPC server returns values for a server item to MATLAB, the OPC server will first convert the value to the COM Variant equivalent of the data type specified by the `daitem` object's DataType property. If the conversion fails, an error message is returned with the value. When the toolbox receives the value, the COM Variant is converted to the equivalent MATLAB data type according to Table 8-1, Conversion from MATLAB Data Type to COM Variant Data Type.

Handling Arrays for Item Values

The OPC Specification supports arrays of values being written to a server item, and read from a server item. However, a specific server item may not accept an array of values. The behavior of the server in that case is server-dependent. For example, one server may use only the first value of the array. Another server may return an error when attempting to write an array of values to a server item that only supports a scalar value. Industrial Communication Toolbox software is not able to determine if a server item accepts only scalar values.

For all of the data types listed in Table 8-1, Conversion from MATLAB Data Type to COM Variant Data Type that can be converted between MATLAB and a COM Variant, scalar and array data are permitted by the toolbox. However, the OPC Specification supports only one-dimensional arrays of data. Higher dimension MATLAB arrays are flattened into a one-dimensional vector when writing data to the OPC server.

Using Events and Callbacks

You can enhance the power and flexibility of your OPC application by using *event callbacks*. An event is a specific occurrence that can happen while an OPC Data Access client object (`opcda` client object) is connected to an OPC server. The toolbox defines a set of events that include starting, stopping, or acquiring records during a logging task, as well as events for asynchronous reads and writes, data changes, and server shutdown notification.

When a particular event occurs, the toolbox can execute a function that you specify. This is called a *callback*. Certain events can result in one or more callbacks. You can use callbacks to perform processing tasks while your client object is connected. For example, you can display a message, analyze data, or perform other tasks. Callbacks are controlled through OPC object properties. Each event type has an associated property. You specify the function that you want executed as the value of that property.

- “Use the Default Callback Function” on page 9-2
- “Event Types” on page 9-4
- “Retrieve Event Information” on page 9-8
- “Create and Execute Callback Functions” on page 9-12

Use the Default Callback Function

In this section...

"Overview to Callback Example" on page 9-2
"Step 1: Create OPC Group Objects" on page 9-2
"Step 2: Configure the Logging Task Properties" on page 9-2
"Step 3: Configure the Callback Properties" on page 9-2
"Step 4: Start the Logging Task" on page 9-3
"Step 5: Clean Up" on page 9-3

Overview to Callback Example

To illustrate how to use callbacks, this section presents a simple example that creates an OPC object hierarchy and associates a callback function with the start event, records acquired event, and stop event of the OPC Data Access Group object (`dagroup` object). For information about all the event callbacks supported by the toolbox, see "Event Types" on page 9-4.

The example uses the default callback function provided with the toolbox, `opccallback`. The default callback function displays the name of the object along with information about the type of event that occurred and when it occurred. To learn how to create your own callback functions, see "Create and Execute Callback Functions" on page 9-12.

Step 1: Create OPC Group Objects

This example creates a hierarchy of OPC objects for the Matrikon Simulation Server. To run this example on your system, you must have the Matrikon Simulation Server installed. Alternatively, you can replace the values used in the creation of the objects with values for a server you can access.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da,'CallbackTest');
itm = additem(grp,{'Random.Real8','Saw-toothed Waves.UInt2'});
```

Step 2: Configure the Logging Task Properties

For this example, we log 20 records at 0.5-second intervals.

```
grp.RecordsToAcquire = 20;
grp.UpdateRate = 0.5;
```

Step 3: Configure the Callback Properties

Set the values of three callback properties. The example uses the default callback function `opccallback`.

```
grp.StartFcn = @opccallback;
grp.StopFcn = @opccallback;
grp.RecordsAcquiredFcn = @opccallback;
```

For this example, specify how often to generate a records acquired event.

```
grp.RecordsAcquiredFcnCount = 5;
```

Step 4: Start the Logging Task

Start the `dagroup` object. The object logs 20 records at 0.5-second intervals, and then stops. With the three callback functions enabled, the object outputs information about each event as it occurs. The records acquired event occurs four times for this example.

```
start(grp)
```

```
OPC Start event occurred at local time 18:52:38
   Group 'CallbackTest': 0 records acquired.
OPC RecordsAcquired event occurred at local time 18:52:41
   Group 'CallbackTest': 5 records acquired.
OPC RecordsAcquired event occurred at local time 18:52:44
   Group 'CallbackTest': 10 records acquired.
OPC RecordsAcquired event occurred at local time 18:52:47
   Group 'CallbackTest': 15 records acquired.
OPC RecordsAcquired event occurred at local time 18:52:49
   Group 'CallbackTest': 20 records acquired.
OPC Stop event occurred at local time 18:52:49
   Group 'CallbackTest': 20 records acquired.
```

Step 5: Clean Up

Always remove toolbox objects from memory, and the variables that reference them, when you no longer need them.

```
disconnect(da)
delete(da)
clear da grp itm
```

Event Types

Industrial Communication Toolbox software supports several different types of events. Each event type has an associated toolbox object property that you can use to specify the function that executes when the event occurs.

The following table lists the supported event types, the name of the object property associated with the event, and a brief description of the event, including the object class associated with the event. For detailed information about these callback properties, see the reference information for the property.

The toolbox generates a specific set of information for each event and stores it in an event structure. To learn more about the contents of these event structures and how to retrieve this information, see “Retrieve Event Information” on page 9-8.

Events and Callback Function Properties

Event	Callback Property	Description
Cancel Async	CancelAsyncFcn	The toolbox generates a cancel async event when an asynchronous operation is cancelled. You cancel an asynchronous operation using the `cancelasync` function. When a cancel async event occurs, the toolbox executes the function specified by the `CancelAsyncFcn` property. By default, the toolbox executes the default callback function for this event, `opccallback`, which displays information about the cancel async event at the MATLAB command line. Cancel async events occur at the `dagroup` object level.
Data Change	DataChangeFcn	The toolbox generates a data change event when the server notifies the toolbox that data for a group has changed. The server will notify the toolbox of data changes only if the group's `Active` property is set to `'on'` and the `Subscription` property is set to `'on'`. For more information on controlling data change events, see “Data Change Events and Subscription” on page 7-8. When a data change event occurs, the toolbox executes the function specified by the `DataChangeFcn` property. Data change events occur at the `dagroup` object level.
Error	ErrorFcn	The toolbox generates an error event when a run-time error occurs, such as a data type conversion error or time-out. Run-time errors do not include configuration errors such as setting an invalid property value. When an error event occurs, the toolbox executes the function specified by the `ErrorFcn` property. By default, the toolbox executes the default callback function for this event, `opccallback`, which displays the error message at the MATLAB command line. Error events occur at the `opcda` client object level.
Read Async	ReadAsyncFcn	The toolbox generates a read async event when an asynchronous read operation completes. You execute an asynchronous read operation using the `readasync` function. When a read async event occurs, the toolbox executes the function specified by the `ReadAsyncFcn` property. By default, the toolbox executes the default callback function for this event, `opccallback`, which displays information about the read async event at the MATLAB command line. Read async events occur at the `dagroup` object level.

Event	Callback Property	Description
Records Acquired	RecordsAcquiredFcn	The toolbox generates a records acquired event every time an integer multiple of a specified number of records have been acquired. You use the `RecordsAcquiredFcnCount` property to specify this number. When a records acquired event occurs, the toolbox executes the function specified by the `RecordsAcquiredFcn` property. Records acquired events occur at the `dagroup` object level.
Shutdown	ShutDownFcn	The toolbox generates a shutdown event when the OPC server notifies the client that the server is about to shut down. When a shutdown event occurs, the toolbox executes the function specified by the `ShutDownFcn` property, and the client object is then disconnected from the server. By default, the toolbox executes the default callback function for this event, `opccallback`, which displays information about the shutdown event at the MATLAB command line. Shutdown events occur at the `opcda` client object level.
Start	StartFcn	The toolbox generates a start event when an object is started. You use the `start` function to start an object. **Note** If an error occurs in the start callback function, the object does not start. When a start event occurs, the toolbox executes the function specified by the `StartFcn` property. Start events occur at the `dagroup` object level.
Stop	StopFcn	The toolbox generates a stop event when the object stops running. An object stops running when the `stop` function is called, or when the specified number of records is acquired. When a stop event occurs, the toolbox executes the function specified by the `StopFcn` property. Stop events occur at the `dagroup` object level.

Event	Callback Property	Description
Timer	TimerFcn	The toolbox generates a timer event when an integer multiple of a specified amount of time expires. You use the `TimerPeriod` property to specify the amount of time. Time is measured relative to when the `opcda` client object is connected. **Note** Some timer events might not execute if your system is significantly slowed or if the `TimerPeriod` is set too small. When a timer event occurs, the toolbox executes the function specified by the `TimerFcn` property. Timer events occur at the `opcda` client object level.
Write Async	WriteAsyncFcn	The toolbox generates a write async event when an asynchronous write operation completes. You execute an asynchronous write operation using the `writeasync` function. When a write async event occurs, the toolbox executes the function specified by the `WriteAsyncFcn` property. By default, the toolbox executes the default callback function for this event, `opccallback`, which displays information about the write async event at the MATLAB command line. Write async events occur at the `dagroup` object level.

Retrieve Event Information

In this section...
"Event Structures" on page 9-8
"Access Data in the Event Log" on page 9-10

Event Structures

Each event has a set of information associated with that event. The information is generated by the OPC server or the toolbox software, and stored in an event structure. This information includes the event type, the time the event occurred, and other event-specific information. For some events, the toolbox records event information in the `opcda` client object's EventLog property. You can also access the event structure associated with an event in a callback function.

For information about accessing event information in a callback function, see "Create and Execute Callback Functions" on page 9-12.

An event structure contains two fields: `Type` and `Data`. For example, this is an event structure for a start event.

```
Type: 'Start'
Data: [1x1 struct]
```

The `Type` field is a character vector that specifies the event type. For a start event, this field contains the value `'Start'`.

The `Data` field is a structure that contains information about the event. The composition of this structure varies, depending on which type of event occurred. For details about the information associated with specific events, see the following sections:

- "Data Fields for Cancel Async, Data Change, Error, Read Async, and Write Async Events" on page 9-8
- "Data Fields for Start, Stop, and Records Acquired Events" on page 9-9
- "Data Fields for Shutdown Events" on page 9-9
- "Data Fields for Timer Events" on page 9-9

Data Fields for Cancel Async, Data Change, Error, Read Async, and Write Async Events

For cancel async, data change, error, read async, and write async events, the `Data` structure contains these fields.

Field Name	Description
`GroupName`	The name of the group associated with the event.
`LocalEventTime`	Absolute time the event occurred, returned in MATLAB date vector format: `[year month day hour minute seconds]`
`TransID`	The transaction ID for the operation. In the case of a cancel async event, `TransID` contains the transaction ID that was cancelled.

Field Name	Description
`Items`	A structure array containing information about each item in the asynchronous operation. The cancel async event structure does not contain this field.

The `Items` structure array for read async events contains the following fields.

Field Name	Description
`ItemID`	The item ID for this record in the structure array.
`Value`	The data value.
`Quality`	The data quality as a character vector.
`TimeStamp`	The time the OPC server updated the value and quality. The time is returned in MATLAB date vector format: `[year month day hour minute seconds]`

The `Items` structure array for write async events contains one field: `ItemID`.

The Items structure array for error events contains the `ItemID` field and an `Error` field, containing a character vector describing the error that occurred for that item.

Data Fields for Start, Stop, and Records Acquired Events

For start, stop, and records acquired events, the `Data` structure contains these fields.

Field Name	Description
`GroupName`	The name of the group associated with the event.
`LocalEventTime`	Absolute time the event occurred, returned in MATLAB date vector format: `[year month day hour minute seconds]`
`RecordsAcquired`	The total number of records acquired in the current logging session.

Data Fields for Shutdown Events

For shutdown events, the `Data` structure contains these fields.

Field Name	Description
`LocalEventTime`	Absolute time the event occurred, returned in MATLAB date vector format: `[year month day hour minute seconds]`
`Reason`	A character vector containing the reason the OPC server provided for shutting down.

Data Fields for Timer Events

For timer events, the `Data` structure contains these fields.

Field Name	Description
`LocalEventTime`	Absolute time the event occurred, returned in MATLAB date vector format: `[year month day hour minute seconds]`

Access Data in the Event Log

While an `opcda` client object is connected, the toolbox stores event information in the `opcda` client object's `EventLog` property. The value of this property is an array of event structures. Each structure represents one event. For detailed information about the composition of an event structure for each type of event, see “Event Structures” on page 9-8.

The toolbox adds event structures to the `EventLog` array in the order in which the events occur. The first event structure reflects the first event recorded, the second event structure reflects the second event recorded, and so on.

Note Data change events, records acquired events, and timer events are not included in the `EventLog`. Event structures for these events (and all the other events) are available to callback functions. For more information, see “Create and Execute Callback Functions” on page 9-12.

To illustrate the event log, this example creates an OPC object hierarchy, executes a logging task, and then examines the object's `EventLog` property:

Step 1: Create the OPC Object Hierarchy

This example creates a hierarchy of OPC objects for the Matrikon Simulation Server. To run this example on your system, you must have the Matrikon Simulation Server installed. Alternatively, you can replace the values used in the creation of the objects with values for a server you can access.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da,'CallbackTest');
itm1 = additem(grp,'Triangle Waves.Real8');
```

Step 2: Start the Logging Task

Start the `dagroup` object. By default, the object acquires 120 records at 0.5-second intervals, and then stops. Wait for the object to stop logging data.

```
start(grp)
wait(grp)
```

Step 3: View the Event Log

Access the `EventLog` property of the `opcda` client object. The execution of the group logging task generated two events: start and stop. Thus the value of the `EventLog` property is a 1-by-2 array of event structures.

```
events = da.EventLog
```

```
events =
1x2 struct array with fields:
    Type
    Data
```

To list the events that are recorded in the `EventLog` property, examine the contents of the `Type` field.

```
{events.Type}
```

```
ans =
    'Start'    'Stop'
```

To get information about a particular event, access the `Data` field in that event structure. The example retrieves information about the stop event.

```
stopdata = events(2).Data
```

```
stopdata =
     LocalEventTime: [2004 3 2 21 33 45.8750]
          GroupName: 'CallbackTest'
    RecordsAcquired: 120
```

Step 4: Clean Up

Always remove toolbox objects from memory, and the variables that reference them, when you no longer need them. Deleting the `opcda` client object also deletes the group and item objects.

```
disconnect(da)
delete(da)
clear da grp itm1
```

Create and Execute Callback Functions

In this section...

"Create Callback Functions" on page 9-12
"Specify Callback Functions" on page 9-13
"View Recently Logged Data" on page 9-15

Create Callback Functions

The power of using event callbacks is that you can perform processing in response to events. You decide which events with which you want to associate callbacks, and which functions these callbacks execute.

Note Callback function execution might be delayed if the callback involves a CPU-intensive task, or if MATLAB software is processing another task.

Callback functions require at least two input arguments:

- The OPC object
- The event structure associated with the event

The function header for this callback function illustrates this basic syntax.

```
function mycallback(obj,event)
```

The first argument, `obj`, is the toolbox object itself. Because the object is available, you can use in your callback function any of the toolbox functions, such as `getdata`, that require the object as an argument. You can also access all object properties, including the parent and children of the object.

The second argument, `event`, is the event structure associated with the event. This event information pertains only to the event that caused the callback function to execute. For a complete list of supported event types and their associated event structures, see "Event Structures" on page 9-8.

In addition to these two required input arguments, you can also specify application-specific arguments for your callback function.

Note If you specify input arguments in addition to the object and event arguments, you must use a cell array when specifying the name of the function as the value of a callback property. For more information, see "Specify Callback Functions" on page 9-13.

Write a Callback Function

This example implements a callback function for a records acquired event. This callback function enables you to monitor the records being acquired by viewing the most recently acquired records in a plot window.

To implement this function, the callback function acquires the last 60 records of data (or fewer if not enough data is available in the toolbox engine) and displays the data in a MATLAB figure window. The function also accesses the event structure passed as an argument to display the time stamp of the event. The `drawnow` command in the callback function forces MATLAB to update the display.

```
function display_opcdata(obj,event)

numRecords = min(obj.RecordsAvailable, 100);
lastRecords = peekdata(obj,numRecords);
[i, v, q, t] = opcstruct2array(lastRecords);
plot(t, v);
isBad = strncmp('Bad', q, 3);
isRep = strncmp('Repeat', q, 6);
hold on
for k=1:length(i)
  h = plot(t(isBad(:,k),k), v(isBad(:,k),k), 'o');
  set(h,'MarkerEdgeColor','k', 'MarkerFaceColor','r')
  h = plot(t(isRep(:,k),k), v(isRep(:,k),k), '*');
  set(h,'MarkerEdgeColor',[0.75, 0.75, 0]);
end
axis tight;
ylim([0, 200]);
datetick('x','keeplimits');
eventTime = event.Data.LocalEventTime;
title(sprintf('Event occurred at %s', ...
  datestr(eventTime, 13)));
drawnow; % force an update of the figure window
hold off;
```

To see how this function can be used as a callback, see “View Recently Logged Data” on page 9-15.

Specify Callback Functions

You associate a callback function with a specific event by setting the value of the OPC object property associated with that event. You can specify the callback function as the value of the property in one of three ways:

- “Use a Character Vector to Specify Callback Functions” on page 9-13
- “Use a Cell Array to Specify Callback Functions” on page 9-14
- “Use Function Handles to Specify Callback Functions” on page 9-14

The following sections provide more information about each of these options.

Note To access the object or event structure passed to the callback function, you must specify the function as a cell array or as a function handle.

Use a Character Vector to Specify Callback Functions

You can specify the callback function as a character vector. For example, this code specifies the callback function `mycallback` as the value of the start event callback property `StartFcn` for the group object `grp`.

```
grp.StartFcn = 'mycallback';
```

In this case, the callback is evaluated in the MATLAB workspace.

Use a Cell Array to Specify Callback Functions

You can specify the callback function as a character vector inside a cell array.

For example, this code specifies the callback function `mycallback` as the value of the start event callback property `StartFcn` for the group object `grp`.

```
grp.StartFcn = {'mycallback'};
```

To specify additional parameters, include them as additional elements in the cell array.

```
time = datestr(now,0);
grp.StartFcn = {'mycallback',time};
```

The first two arguments passed to the callback function are still the OPC object (`obj`) and the event structure (`event`). Additional arguments follow these two arguments.

Use Function Handles to Specify Callback Functions

You can specify the callback function as a function handle.

For example, this code specifies the callback function `mycallback` as the value of the start event callback property `StartFcn` for the group object `grp`.

```
grp.StartFcn = @mycallback;
```

To specify additional parameters, include the function handle and the parameters as elements in the cell array.

```
time = datestr(now,0);
grp.StartFcn = {@mycallback,time};
```

If you are executing a local callback function from within a file, you must specify the callback as a function handle.

Specify a Toolbox Function as a Callback

In addition to specifying callback functions of your own creation, you can also specify toolbox functions as callbacks. For example, this code sets the value of the stop event callback to the `start` function.

```
grp.StopFcn = @start;
```

Disable Callbacks

If an error occurs in the execution of the callback function, the toolbox disables the callback and displays a message similar to the following.

```
start(grp)
```

```
??? Error using ==> myrecords_cb
Too many input arguments.

Warning: The RecordsAcquiredFcn callback is being disabled.
```

To enable a callback that has been disabled, set the value of the property associated with the callback.

View Recently Logged Data

This example configures an OPC object hierarchy and sets the records acquired event callback function property to the `display_opcdata` function, created in "Write a Callback Function" on page 9-12.

When run, the example displays the last 60 records of acquired data every time 5 records have been acquired. Repeat values are highlighted with magenta circles, and bad values are highlighted with red circles.

Step 1: Create the OPC Object Hierarchy

This example creates a hierarchy of OPC objects for the Matrikon Simulation Server. To run this example on your system, you must have the Matrikon Simulation Server installed. Alternatively, you can replace the values used in the creation of the objects with values for a server you can access.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da)
grp = addgroup(da,'CallbackTest');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-toothed Waves.UInt2');
```

Step 2: Configure Property Values

This example sets the `UpdateRate` value to 0.2 seconds, and the `RecordsToAcquire` property to 200. The example also specifies as the value of the `RecordsAcquiredFcn` callback the event callback function `display_opcdata`, created in "Write a Callback Function" on page 9-12. The object will execute the `RecordsAcquiredFcn` every 5 records, as specified by the value of the `RecordsAcquiredFcnCount` property.

```
grp.UpdateRate = 0.2;
grp.RecordsToAcquire = 200;
grp.RecordsAcquiredFcnCount = 5;
grp.RecordsAcquiredFcn = @display_opcdata;
```

Step 3: Acquire Data

Start the `dagroup` object. Every time 5 records are acquired, the object executes the `display_opcdata` callback function. This callback function displays the most recently acquired records logged to the memory buffer.

```
start(grp)
wait(grp)
```

Step 4: Clean Up

Always remove toolbox objects from memory, and the variables that reference them, when you no longer need them. Deleting the `opcda` client object also deletes the group and item objects.

```
disconnect(da)
delete(da)
clear da grp itm1 itm2
```

Using the OPC Block Library

- “Block Library Overview” on page 10-2
- “Read and Write Data from a Model” on page 10-3
- “Use the OPC Client Manager” on page 10-11

Block Library Overview

Industrial Communication Toolbox software includes a Simulink interface called the OPC block library. This library is a tool for sending data from your Simulink model to an OPC server, or querying an OPC server to receive live data into your model. You use blocks from the OPC block library with blocks from other Simulink libraries to create models capable of sophisticated OPC server communications.

The OPC block library requires Simulink, a tool for simulating dynamic systems. Simulink is a model definition environment. Use Simulink blocks to create a block diagram that represents the computations of your system or application. Simulink is also a model simulation environment in which you can see how your system behaves.

The best way to learn about the OPC block library is to observe an example, such as “Read and Write Data from a Model” on page 10-3.

Read and Write Data from a Model

In this section...

"Example Overview" on page 10-3

"Step 1: Create New Model in Simulink Editor" on page 10-3

"Step 2: Open the OPC Block Library" on page 10-3

"Step 3: Drag OPC Blocks into the Editor" on page 10-4

"Step 4: Drag Other Blocks to Complete the Model" on page 10-5

"Step 5: Configure OPC Servers for the Model" on page 10-5

"Step 6: Specify the Block Parameter Values" on page 10-7

"Step 7: Connect the Blocks" on page 10-9

"Step 8: Run the Simulation" on page 10-10

Example Overview

This section provides a step-by-step example to illustrate how to use the OPC block library. The example builds a simple model using the blocks in the OPC block library with blocks from other Simulink libraries.

This example writes a sine wave to the Matrikon OPC Simulation Server, and reads the data back from the same server. You use the OPC Write block to send data to the OPC server, and the OPC Read block to read that same data back into your model.

Note To run the code in the following examples, you must have the Matrikon OPC Simulation Server available on your local machine. For information on installing this, see "Install an OPC DA or HDA Simulation Server for OPC Classic Examples" on page 1-14. The code used in this example requires only minor changes to work with other servers.

Step 1: Create New Model in Simulink Editor

1. To start Simulink and create a new model, enter the following at the MATLAB command prompt:

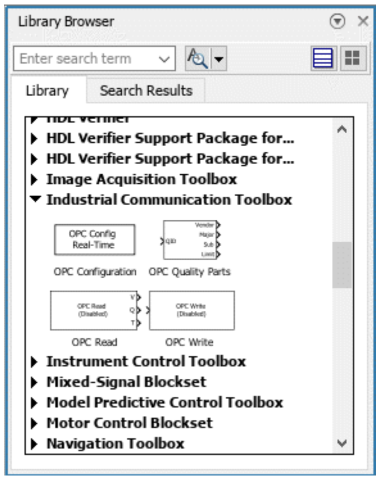
   ```
   simulink
   ```

   In the Simulink start page dialog, click **Blank Model**, and then **Create Model**. An empty, Editor window opens.
2. In the Editor, click **File > Save As** to assign a name to your new model.

Step 2: Open the OPC Block Library

1. In the model Editor window, click **Library Browser**.

   The Simulink Library Browser opens in the left pane of the Editor, with a tree of available block libraries in alphabetical order.
2. Expand the `Industrial Communication Toolbox` node.



Alternatively, you can open the OPC block library in a standalone window by typing the following command at the MATLAB command prompt:

```
opclib
```

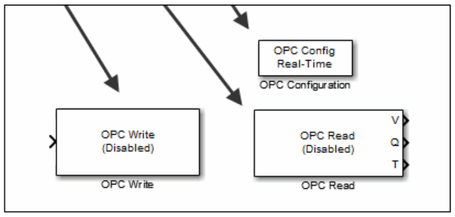
Step 3: Drag OPC Blocks into the Editor

The OPC block library contains four blocks

- OPC Configuration
- OPC Quality Parts
- OPC Read
- OPC Write

You can use these blocks to configure and manage connections to servers, to send and receive live data between your OPC server and your simulation, and to analyze OPC quality.

To use the blocks in a model, select each block in the library and drag the block into the Simulink Editor. For this example, you need one instance each of the OPC Configuration, OPC Write, and OPC Read block in your model.

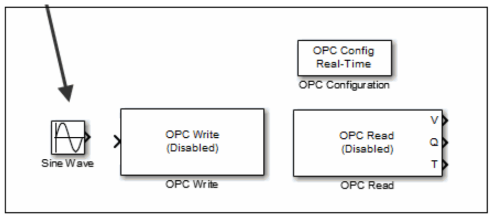


Note Block names are not shown by default in the model. To display the hidden block names while working in the model, select **Display** and clear the **Hide Automatic Names** check box.

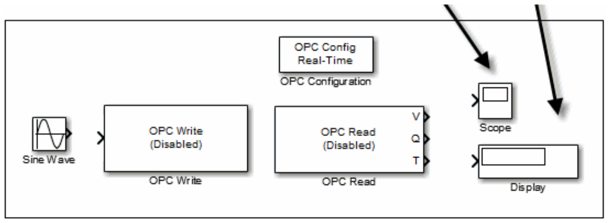
Step 4: Drag Other Blocks to Complete the Model

Your model requires three more blocks. One block provides the data sent to the server; the other two blocks display the data received from the server.

To send a sine wave to the server, you can use the Sine Wave block. To access the Sine Wave block, expand the Simulink node in the browser tree, and click the Sources library entry. From the blocks displayed in the right pane, drag the Sine Wave block into the Simulink Editor and place it to the left of the OPC Write block.



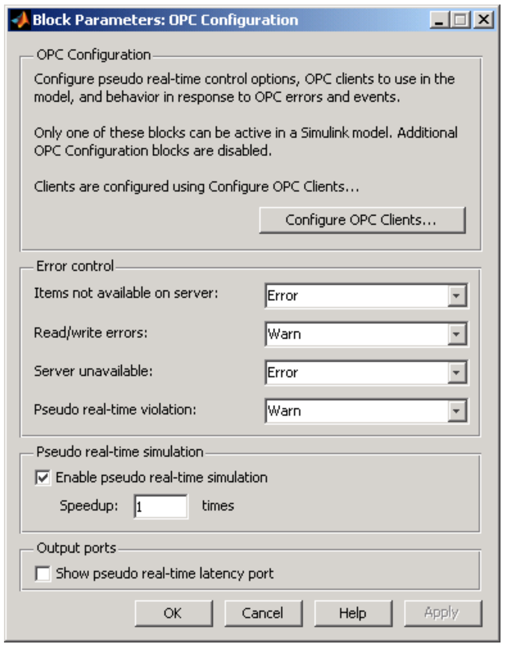
You can use the Scope block to show the value received from the server, and a Display block to view the quality of the item. (You will remove the time stamp output port in the next step.) To access the Scope block, click the `Sinks` library entry in the expanded Simulink node in the browser tree. From the blocks displayed in the right pane, drag the Scope block into the Simulink Editor and place it above and to the right of the OPC Read block. Also drag a Display block into the Simulink Editor and place it below the Scope block.



Step 5: Configure OPC Servers for the Model

To communicate with OPC servers from Simulink, you first need to configure those servers in the model. The OPC Configuration block manages and configures OPC servers for a Simulink model. Each OPC Read or OPC Write block uses one server from the configured servers, and defines the items to read from or write to.

1 Double-click the OPC Configuration block to open its parameters dialog.


Block Parameters: OPC Configuration

OPC Configuration

Configure pseudo real-time control options, OPC clients to use in the model, and behavior in response to OPC errors and events.

Only one of these blocks can be active in a Simulink model. Additional OPC Configuration blocks are disabled.

Clients are configured using Configure OPC Clients...

Error control

Items not available on server:	Error
Read/write errors:	Warn
Server unavailable:	Error
Pseudo real-time violation:	Warn

Pseudo real-time simulation

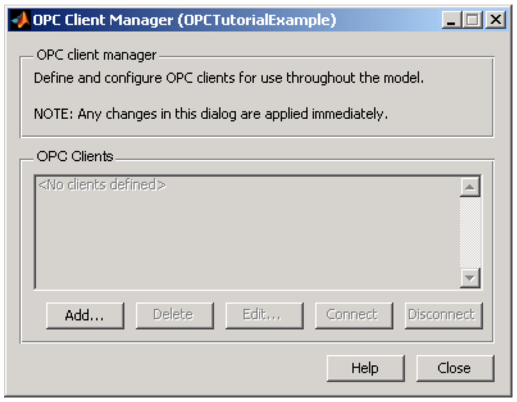
[✓] Enable pseudo real-time simulation

Speedup: 1 times

Output ports

[ ] Show pseudo real-time latency port

2 Click **Configure OPC Clients** to open the OPC Client Manager.


OPC Client Manager (OPCTutorialExample)

OPC client manager

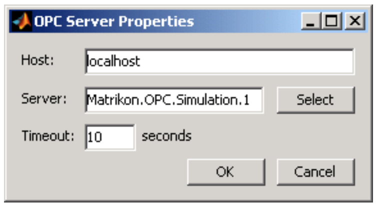
Define and configure OPC clients for use throughout the model.

NOTE: Any changes in this dialog are applied immediately.

OPC Clients

<No clients defined>

3 Click **Add** to open the OPC Server Properties dialog. Specify the ID of the server as `'Matrikon.OPC.Simulation.1'` (or click **Select** and choose the server from the list of available OPC servers).

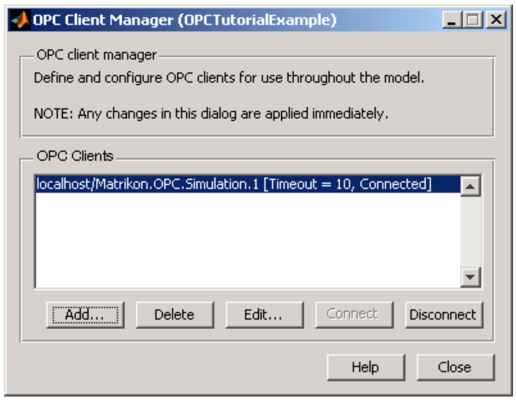

OPC Server Properties

Host: localhost

Server: Matrikon.OPC.Simulation.1

Timeout: 10 seconds

4 Click **OK** to add the OPC server to the OPC Client Manager.


OPC Client Manager (OPCTutorialExample)

OPC client manager

Define and configure OPC clients for use throughout the model.

NOTE: Any changes in this dialog are applied immediately.

OPC Clients

localhost/Matrikon.OPC.Simulation.1 [Timeout = 10, Connected]

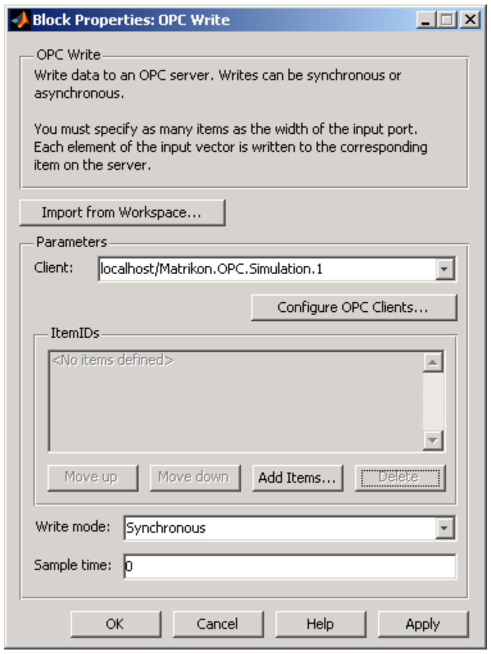
The Matrikon OPC Simulation Server is now available throughout the model for reading and writing.

5 Your model will use default values for all other settings in the OPC Configuration block. Click **OK** in the OPC Configuration dialog to close that dialog.

Step 6: Specify the Block Parameter Values

You set parameters for the blocks in your model by double-clicking on each block.

1 Double-click the OPC Write block to open its parameters dialog. The Matrikon server is automatically selected for you as the OPC client to use in this block. You need to specify the items for writing.


Block Properties: OPC Write

OPC Write

Write data to an OPC server. Writes can be synchronous or asynchronous.

You must specify as many items as the width of the input port. Each element of the input vector is written to the corresponding item on the server.

Parameters

Client: localhost/Matrikon.OPC.Simulation.1

ItemIDs

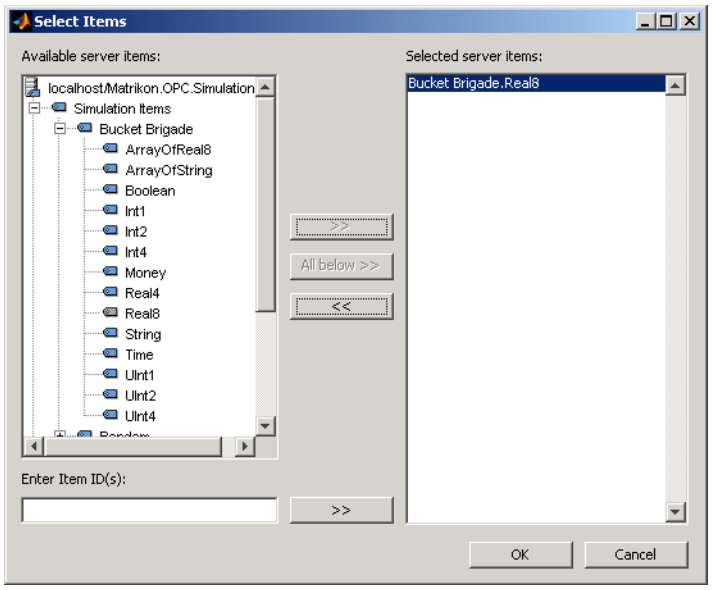
<No items defined>

Write mode: Synchronous

Sample time: 0

2 Click **Add Items** to display a name space browser for the Matrikon OPC Simulation Server.

3 Expand the Simulation Items node in the name space, then expand the Bucket Brigade node. Select the Real8 node and click **>>** to add that item to the selected items list.



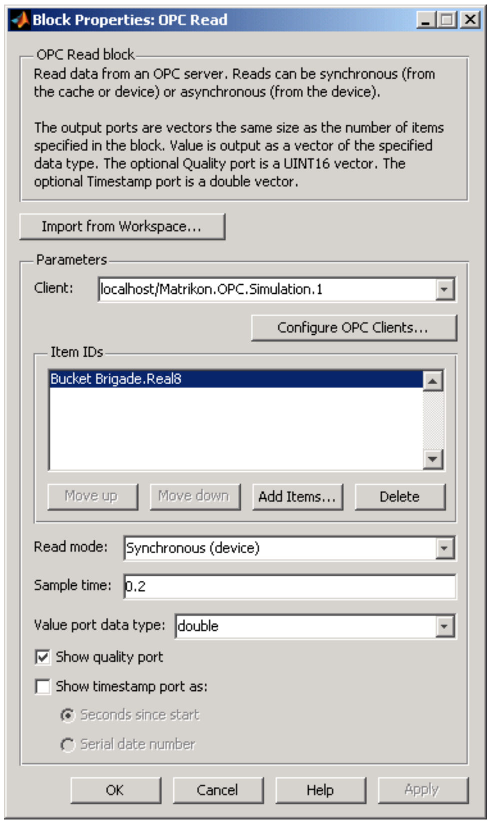
4 Click **OK** to add the item `Bucket Brigade.Real8` to the OPC Write block's ItemIDs list.

5 In the OPC Write parameters dialog, click **OK** to accept the changes and close the dialog.

6 Double-click the OPC Read block to open its dialog. Add the same item to the OPC Read block, repeating steps 2–5 that you followed for the OPC Write block in this section.

7 Set the read mode to `'Synchronous (device)'` and the sample time for the block to `0.2`.

8 Also uncheck the `'Show timestamp port'` option. This step removes the time stamp output port from the OPC Read block.



Block Properties: OPC Read

OPC Read block

Read data from an OPC server. Reads can be synchronous (from the cache or device) or asynchronous (from the device).

The output ports are vectors the same size as the number of items specified in the block. Value is output as a vector of the specified data type. The optional Quality port is a UINT16 vector. The optional Timestamp port is a double vector.

Import from Workspace...

Parameters

Client: localhost/Matrikon.OPC.Simulation.1

Configure OPC Clients...

Item IDs

Bucket Brigade.Real8

Move up | Move down | Add Items... | Delete

Read mode: Synchronous (device)

Sample time: 0.2

Value port data type: double

☑ Show quality port

☐ Show timestamp port as:

● Seconds since start

○ Serial date number

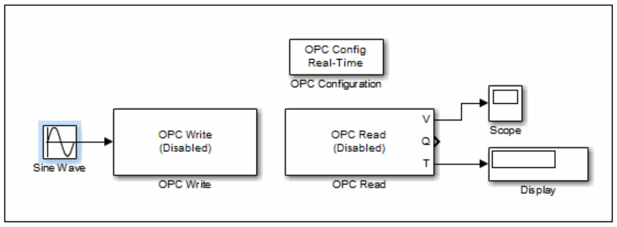
OK | Cancel | Help | Apply

Step 7: Connect the Blocks

Make a connection between the Sine Wave block and the OPC Write block. When you move the cursor near the output port of the Sine Wave block, the cursor becomes crosshairs. Click the Sine Wave output port and hold the mouse button; drag to the input port of the OPC Write block, and release the button.

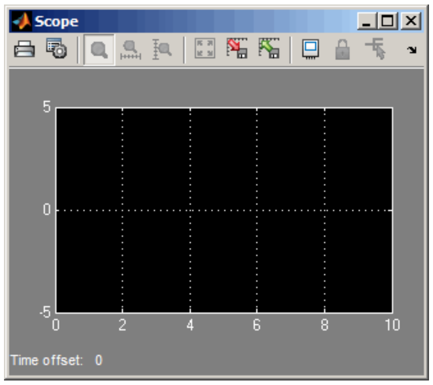
In the same way, make a connection between the first output port of the OPC Read block (labeled V) and the input port of the Scope block. Then connect the other output port of the OPC Read block (labeled Q) to the input port of the Display block.

Note that the OPC Write and OPC Read blocks do not directly connect together within the model. The only communication between them is through an item on the server, which you defined in “Step 5: Configure OPC Servers for the Model” on page 10-5.



Step 8: Run the Simulation

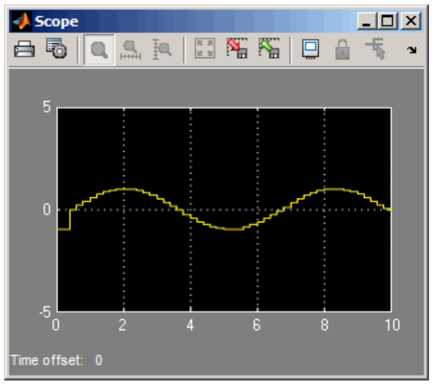
Before you run the simulation, double-click the Scope block to open the scope view.



To run the simulation, click **Run** in the Simulink Editor toolstrip.

The model writes a sine wave to the OPC server, reads back from the server, and displays the wave in the scope trace. In addition, the quality value is set to 192, which indicates a good quality (see “OPC Quality” on page A-2).

While the simulation is running, the status bar at the bottom of the model window updates the progress of the simulation, and the sine wave is displayed in the Scope window.



Use the OPC Client Manager

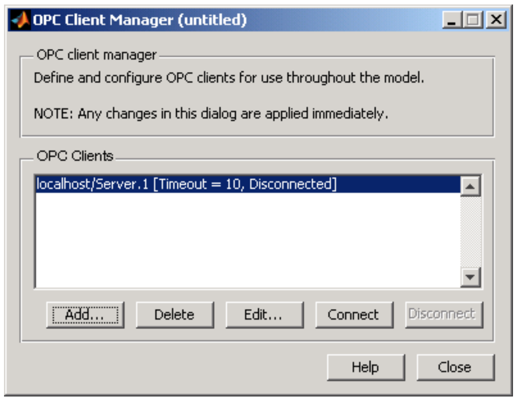
In this section...

"Introduction to the OPC Client Manager" on page 10-11
"Add Clients to the OPC Client Manager" on page 10-11
"Remove Clients from the OPC Client Manager" on page 10-12
"Modify the Server Timeout Value for a Client" on page 10-12
"Control Client/Server Connections" on page 10-12

Introduction to the OPC Client Manager

The OPC Client Manager displays and manages all clients for a Simulink model. Using the OPC Client Manager, you associate one or more clients with a particular model. Each time you use an OPC Read or OPC Write block, you choose the client for that block from the list of configured clients. By defining a single list of clients in the OPC Client Manager, you enable a Simulink model to reuse clients among OPC Read and OPC Write blocks.

You access the OPC Client Manager from the parameters dialog of the OPC Configuration, OPC Read, or OPC Write block, by clicking **Configure OPC Clients**. A dialog similar to the following figure appears.



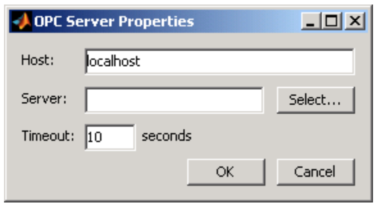
Define and configure OPC clients for use throughout the model.

NOTE: Any changes in this dialog are applied immediately.

localhost/Server.1 [Timeout = 10, Disconnected]

Add Clients to the OPC Client Manager

You add clients to the OPC Client Manager by clicking **Add**. The following dialog box appears.



Host: localhost
Server:
Timeout: 10

Specify the host in the **Host** edit box. You can then type the Server ID of the required server, or use **Select** to query the host for a list of servers.

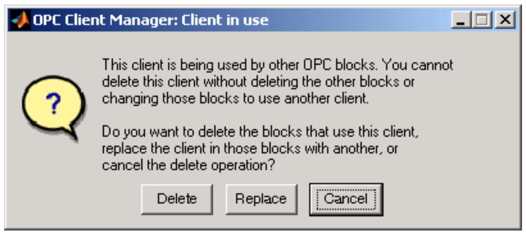
Specify the timeout (in seconds) to use when communicating with the server.

When you click **OK**, the client is added to the OPC Clients list in the OPC Client Manager. You can now use that client in one or more OPC Read or OPC Write blocks within that model.

Remove Clients from the OPC Client Manager

To remove a client from the OPC Client Manager, select the client in the **OPC Clients** list and click **Delete**. A confirmation dialog appears. Click **Delete** to remove the client from the OPC Client Manager.

If you attempt to remove a client that is referenced by one or more OPC library blocks, you see the following dialog.


OPC Client Manager: Client in use

This client is being used by other OPC blocks. You cannot delete this client without deleting the other blocks or changing those blocks to use another client.

Do you want to delete the blocks that use this client, replace the client in those blocks with another, or cancel the delete operation?

Click **Delete** to remove all blocks that reference the client you want to delete.

Click **Replace** to replace the referenced client with another client in the OPC Client list (this choice is available only if another client is available), and select the replacement client from the resulting list. Click **Cancel** to cancel the delete operation.

Modify the Server Timeout Value for a Client

Click **Edit** to modify the timeout property of the selected client. The timeout value is specified in seconds, and applies to all server operations (connect, disconnect, read, write).

Control Client/Server Connections

Industrial Communication Toolbox software automatically attempts to connect a client configured in the OPC Client Manager to its server. This enables you to browse the server name space for items, and speeds up the initialization process of simulating a model.

You can control the client connection status by highlighting a client in the **OPC Client** list and clicking **Connect** or **Disconnect**.

The OPC block library automatically reconnects any disconnected client to its server when you run a simulation.

Properties

opcda Object Properties

Configure OPC DA client

Description

Use the properties of the `opcda` client object to access server connection parameters and specify other high level behaviors.

Properties

General Settings

`EventLog` — Event information log
structure

This property is read-only.

`EventLog` contains a structure array that stores information related to Industrial Communication Toolbox software events. Every element in the structure array corresponds to an event.

Each element in the `EventLog` structure contains the fields `Type` and `Data`. The `Type` value can be `'WriteAsync'`, `'ReadAsync'`, `'CancelAsync'`, `'Shutdown'`, `'Start'`, `'Stop'`, or `'Error'`.

`Data` stores event-specific information as a structure. For information on the fields contained in `Data`, refer to the associated callback property reference pages. For example, to find information on the fields contained in Data for a `Start` event, refer to the StartFcn property.

You specify the maximum number of events to store with the EventLogMax property.

Note that some events are not stored in the `EventLog`. If you want to store these events, you must specify a callback for that event.

You can execute a callback function when an event occurs by specifying a function for the associated callback property. For example, to execute a callback when a read async event is generated, you use the ReadAsyncFcn property.

If the event log is full (the number of events in the log equals the value of the EventLogMax property) and a new event is received, the oldest event is removed to make space for the new event. You clear the event log using the `cleareventlog` function.

Example

The following example creates a client and configures a group with two items. A 30-second logging task is run, and after 10 seconds the item values are read. When the logging task stops, the event log is retrieved and examined.

```
da = opcda('localhost', 'Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da, 'EvtLogExample');
itm1 = additem(grp, 'Random.Real8');
itm2 = additem(grp, 'Triangle Waves.UInt1');
```

```
set(grp, 'UpdateRate', 1, 'RecordsToAcquire', 30);
start(grp);
pause(10);
tid = readasync(grp);
wait(grp);
el = get(da, 'EventLog')
el = get(da, 'EventLog')

el =
1x3 struct array with fields:
    Type
    Data
```

Now examine the first event, which is the start event.

```
el(1)
ans =
    Type: 'Start'
    Data: [1x1 struct]
```

The Data field contains the following information.

```
el(1).Data
ans =
     LocalEventTime: [2004 1 13 16 16 25.1790]
          GroupName: 'EvtLogExample'
    RecordsAcquired: 0
```

The second event is a `ReadAsync` event. Examine the `Data` structure and the first element of the `Items` structure.

```
el(2)
ans =
    Type: 'ReadAsync'
    Data: [1x1 struct]

el(2).Data
ans =
    LocalEventTime: [2004 1 13 16 16 35.2100]
           TransID: 2
         GroupName: 'EvtLogExample'
             Items: [2x1 struct]

el(2).Data.Items(1)
ans =
       ItemID: 'Random.Real8'
        Value: 2.4619e+003
      Quality: 'Good: Non-specific'
    TimeStamp: [2004 1 13 16 16 35.1870]
```

Data Types: `struct`

EventLogMax — Maximum number of events to store in event log
1000 (default) | double

If the event log is full (the number of events in the log equals the value of the `EventLogMax` property) and a new event is received, the oldest event is removed to make space for the new event. You clear the event log using the `cleareventlog` function.

By default, `EventLogMax` is set to `1000`. To continually store events, specify a value of `Inf`. To store no events, specify a value of `0`. If `EventLogMax` is reduced to a value less than the number of existing events in the event log, the oldest events are removed until the number of events is equal to `EventLogMax`.

Example: `1000`

Data Types: `double`

`Group` — Data Access Group objects contained by client
`dagroup` array

This property is read-only.

`Group` is a vector of `dagroup` objects contained by the `opcda` object. `Group` is initially an empty vector. The size of `Group` increases as you add groups with the `addgroup` function, and decreases as you remove groups with the `delete` function.

`Host` — DNS name or IP address of server
char

This property is read-only.

`Host` is the name or IP address of the machine hosting the OPC server. If you specify the host using an IP address, no name resolution is performed on that address.

Data Types: `char`

`Name` — Descriptive name for OPC DA client object
char

The default object creation behavior is to automatically assign a name to all objects. For the `opcda` object, `Name` follows the naming scheme `'Host/ServerID'`. For the `dagroup` object, if a name is not specified upon creation, the name returned by the OPC server is used, or a unique name is automatically assigned to the group. Automatically assigned group names follow the naming scheme `'groupN'` where `N` is an integer.

You can change the `Name` of an object at any time. The `Name` can be any character vector, and is used for display and identification purposes only.

Data Types: `char`

`Server ID` — Server identity
char

`ServerID` is the COM style program ID that the `opcda` object connects to. The program ID is normally defined during installation of the OPC server.

Use `opcserverinfo` to find a list of available servers and their server IDs.

Data Types: `char`

`Status` — Status of connection to OPC server
`'disconnected'` (default) | `'connected'`

This property is read-only.

`Status` can be `'disconnected'` or `'connected'`. You connect an `opcda` object with the `connect` function and disconnect with the `disconnect` function. If the `opcda` object is connected to a server and the server shuts down, the `Status` property is set to `'disconnected'`.

Example: `'connected'`

Data Types: `char`

Tag — Label to associate with OPC object
char

You configure `Tag` to be a character vector value that uniquely identifies an OPC object.

`Tag` is particularly useful when constructing programs that would otherwise need to define the toolbox object as a global variable, or pass the object as an argument between callback routines. You can return a toolbox object with the `opcfind` function by specifying the `Tag` property value.

Data Types: `char`

`Timeout` — Maximum time to wait for completion of instruction to server
`10` (default) | double

You configure `Timeout` to be the maximum time, in seconds, to wait for completion of a synchronous read or a synchronous write operation. If a timeout occurs, the read or write operation aborts. You can set the property to any value in the range `[0 Inf]`. The default value is `10`.

You can use `Timeout` to abort functions that block access to the MATLAB command line.

For asynchronous read or write operations, `Timeout` specifies the time to wait for the server to acknowledge the request. It does not limit the time for the instruction to be completed by the server.

Example: `60`

Data Types: `double`

Type — OPC object type
char

This property is read-only.

`Type` indicates the type of the object. The OPC object types are `'opcda'`, `'dagroup'`, and `'daitem'`. Once an object is created, the value of `Type` is automatically defined, and cannot be changed.

You can identify OPC objects of a given type using the `opcfind` function and the `Type` value.

Example: `'opcda'`

Data Types: `char`

`UserData` — Data to associate with OPC object
any type

You can configure `UserData` to store data that you want to associate with an OPC object. The object does not use this data directly, but you can access it using the `get` function.

Callback Function Settings

`ErrorFcn` — Callback function file to execute when error event occurs

function handle | char | cell

You configure `ErrorFcn` to execute a callback function file when an error event occurs. An error event is generated when an asynchronous transaction fails. For example, an asynchronous read on items that cannot be read generates an error event. An error event is not generated for configuration errors such as setting an invalid property value, nor for synchronous read and write operations.

When an `Error` event occurs, the function specified in `ErrorFcn` is passed two parameters: `Obj` and `EventInfo`. `Obj` is the object associated with the event, and `EventInfo` is an event structure containing the fields `Type` and `Data`. The `Type` field is set to `'Error'`. The `Data` field contains a structure with the following fields:

Field Name	Description
`LocalEventTime`	The local time (as a date vector) the event occurred.
`TransID`	The transaction ID associated with the event.
`GroupName`	The group name.
`Items`	A structure containing information on each item that generated an error during that transaction.

The `Items` structure array contains the following fields:

Field Name	Description
`ItemID`	The item name.
`Error`	The error message.

The default value for `ErrorFcn` is `@opccallback`.

Note that error event information is also stored in the EventLog property.

Example: `@opccallback`

Data Types: `char` | `cell` | `function_handle`

`ShutDownFcn` — Callback function file to execute when OPC server shuts down

function handle | char | cell

You configure `ShutDownFcn` to execute a callback function file when the OPC server shuts down. Prior to calling the `ShutDownFcn` callback, the Status property of the `opcda` object is changed to `'disconnected'`.

When a shutdown event occurs, the function specified in `ShutDownFcn` is passed two parameters: `Obj` and `EventInfo`. `Obj` is the object associated with the event, and `EventInfo` is an event structure containing the fields `Type` and `Data`. The `Type` field is set to `'Shutdown'`. The `Data` field contains a structure with the following fields.

Field Name	Description
`LocalEventTime`	The time the event occurred, as a MATLAB date vector.
`Reason`	The reason for the server shutdown.

Shutdown event information is stored in the `EventLog` property.

Example: `@opccallback`

Data Types: `char` | `cell` | `function_handle`

`TimerFcn` — Callback function file to execute when predefined period passes
function handle | char | cell

You configure `TimerFcn` to execute a callback function file when a timer event occurs. A timer event occurs when the time specified by the TimerPeriod property passes. Timer events are only generated when the Status property is set to `'connected'`. Timer events will stop being generated when the object's Status is set to `'disconnected'`, either by a `disconnect` function call, or when the server shuts down.

Some timer events may not be processed if your system is significantly slowed or if the TimerPeriod value is too small. Timer event information is not stored in the EventLog property.

Example: `@timercallback`

Data Types: `char` | `cell` | `function_handle`

`TimerPeriod` — Period between timer events
`10` (default) | double

`TimerPeriod` specifies the time, in seconds, that must pass before the callback function specified by TimerFcn is called. The setting can be any value in the range `[0.001 Inf]`. The default value is `10`.

Some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value is too small.

Example: `1`

Data Types: `double`

Version History

Introduced before R2006a

See Also

Properties
dagroup Object Properies Properties | daitem Object Properties Properties

Functions
`opcda`

dagroup Object Properties

Configure OPC `dagroup` object

Description

Use the properties of the `dagroup` object to control reading, writing, logging, and so on.

Properties

General Settings

`DeadbandPercent` — Percentage change in item value that causes subscription callback
`0` (default) | `0-100`

You configure `DeadbandPercent` to a value between `0` and `100`. The default value is `0`, which specifies that any value change will update the OPC server's cache. A non-zero value results in the cache value being updated only if the difference between the cached value and the current value of the item exceeds:

`DeadbandPercent` * (`High EU` - `Low EU`) / 100 (11-1)

The `DeadbandPercent` property affects only items that have an analog data type and `'High EU'` and `'Low EU'` properties defined (property IDs `102` and `103`, respectively). You can query data types and item properties using `serveritemprops`.

Note OPC servers might not implement the `DeadbandPercent` property behavior, even for values that have `High EU` and `Low EU` properties defined. For servers that do not support `DeadbandPercent`, an error is generated if you attempt to set the `DeadbandPercent` property to a value other than `0`.

`DeadbandPercent` is applied group-wide for all analog `daitem` objects, and is used to prevent noisy signals from updating the client unnecessarily.

Example: `10`

Data Types: `double`

`GroupType` — Public status of dagroup object
`'private'` (default) | `'public'`

This property is read-only.

`GroupType` indicates whether a group is `private` or `public`. A private group is local to the `opcda` client, and other clients must create their own private groups. A public group is available from the server for other OPC clients on the network.

Example: `'private'`

Data Types: `char`

`Item` — Data access item objects contained in group
array of `daitem` objects

This property is read-only.

`Item` is a vector of `daitem` objects contained in the `dagroup` object. `Item` is initially an empty vector. The size of `Item` increases as you add items with the `additem` function, and decreases as you remove items with the `delete` function.

Data Types: `daitem`

`Name` — Descriptive name for OPC DA group object
char

The default object creation behavior is to automatically assign a name to all objects. For the `opcda` object, `Name` follows the naming scheme `'Host/ServerID'`. For the `dagroup` object, if a name is not specified upon creation, the name returned by the OPC server is used, or a unique name is automatically assigned to the group. Automatically assigned group names follow the naming scheme `'groupN'` where `N` is an integer.

You can change the `Name` of an object at any time. The `Name` can be any character vector, and is used for display and identification purposes only.

Data Types: `char`

`Parent` — OPC object that contains this dagroup object
OPC DA client object

This property is read-only.

For `dagroup` objects, `Parent` indicates the `opcda` client object that contains the group.

Data Types: `DA client object`

`Tag` — Label to associate with OPC object
char

You configure `Tag` to be a character vector value that uniquely identifies an OPC object.

`Tag` is particularly useful when constructing programs that would otherwise need to define the toolbox object as a global variable, or pass the object as an argument between callback routines. You can return a toolbox object with the `opcfind` function by specifying the `Tag` property value.

Data Types: `char`

`TimeBias` — Time bias of group
`0` (default) | double

This property is read-only.

`TimeBias` indicates the time difference between the server and client machines. In some cases the data might have been collected by a device operating in a time zone other than that of the client. Then it is useful to know what the time of the device was when the data was collected (e.g., to determine what shift was on duty at the time).

The time is indicated in minutes, and can be positive or negative.

Example: `60`

Data Types: `double`

Type — OPC object type
char

This property is read-only.

`Type` indicates the type of the object. The OPC object types are `'opcda'`, `'dagroup'`, and `'daitem'`. Once an object is created, the value of `Type` is automatically defined, and cannot be changed.

You can identify OPC objects of a given type using the `opcfind` function and the `Type` value.

Example: `'dagroup'`

Data Types: `char`

`UpdateRate` — Rate, in seconds, at which subscription callbacks occur
`0.5` (default) | double

`UpdateRate` specifies the rate, in seconds, at which subscription callbacks occur. This determines how often the cached data can be updated and how often data change events can occur. Consequently, `UpdateRate` also controls the rate at which data is logged. You start logging data change events with the `start` function.

Data change events can occur only for active items in an active group. Additionally, subscription must be enabled for the group.

Servers can select an update rate that differs from the requested value. If this occurs, `UpdateRate` is automatically updated with the returned value. By specifying an update rate of `0`, updates will occur as soon as new information becomes available for the `daitem` object. New information is considered to be a change in the Quality property, or a change in the data Value that exceeds the DeadbandPercent property value.

Example: `1.0`

Data Types: `double`

`UserData` — Data to associate with OPC object
any type

You can configure `UserData` to store data that you want to associate with an OPC object. The object does not use this data directly, but you can access it using the `get` function.

Callback Function Settings

`CancelAsyncFcn` — Callback function to execute when asynchronous operation is canceled
@opccallback (default) | function handle | char | cell

You configure `CancelAsyncFcn` to execute a callback function when a cancel async event occurs. A cancel async event occurs after an asynchronous read or write operation is canceled.

When a cancel async event occurs, the function specified in `CancelAsyncFcn` is passed two parameters: `Obj` and `EventInfo`. `Obj` is the object associated with the event, and `EventInfo` is an event structure containing the fields `Type` and `Data`. The `Type` field is set to `'CancelAsync'`. The `Data` field contains a structure with the fields shown below.

Field Name	**Description**
`LocalEventTime`	The time, as a MATLAB date vector, that the event occurred.
`TransID`	The transaction ID of the canceled read or write asynchronous operation.
`GroupName`	The group name.

Cancel async event information is stored in the EventLog property.

Example: `@opccallback`

Data Types: `char` | `cell` | `function_handle`

`DataChangeFcn` — Callback function to execute when data change event occurs
function handle | char | cell

You configure `DataChangeFcn` to execute a callback function when a data change event occurs. A data change event occurs for subscribed active items within an active group when the value or quality of the item has changed. The events will happen no faster than the time specified for the UpdateRate property of the group. The DeadbandPercent property is used to determine what percentage change in the value or quality initiates the callback. A data change event is only generated when both the Active and Subscription properties are `'on'`.

When a data change event occurs, the function specified in `DataChangeFcn` is passed two parameters: `Obj` and `EventInfo`. `Obj` is the object associated with the event, and `EventInfo` is an event structure containing the fields `Type` and `Data`. The `Type` field is set to `'DataChange'`. The `Data` field contains a structure with the fields defined below.

Field Name	**Description**
`LocalEventTime`	The time, as a MATLAB date vector, that the event occurred
`TransID`	`0`, or the `Refresh` transaction ID if the data change event was generated by `refresh`
`GroupName`	The group name
`Items`	A structure containing information about each item whose value or quality updated

The `Items` structure contains the fields defined below.

Field Name	**Description**
`ItemID`	The item name
`Value`	The data value
`TimeStamp`	The time, as a MATLAB date vector, that the server's cache was updated

Data change event information is not stored in the `EventLog` property

Example: `@readNewData`

Data Types: `char` | `cell` | `function_handle`

`ReadAsyncFcn` — Callback function to execute when asynchronous read completes
`@opccallback` (default) | function handle | char | cell

You configure `ReadAsyncFcn` to execute a callback function when an asynchronous read operation completes. You execute an asynchronous read with the `readasync` function. A read async event occurs immediately after the data is returned by the server to the MATLAB workspace.

When a read async event occurs, the function specified in `ReadAsyncFcn` is passed two parameters: `Obj` and `EventInfo`. `Obj` is the object associated with the event, and `EventInfo` is an event structure containing the fields `Type` and `Data`. The `Type` field is set to `'ReadAsync'`. The `Data` field contains a structure with the fields defined below.

Field Name	Description
`LocalEventTime`	The time, as a MATLAB date vector, that the event occurred.
`TransID`	The transaction ID for the asynchronous read operation.
`GroupName`	The group name.
`Items`	A structure containing information about each item whose value or quality updated.

The `Items` structure contains the fields defined below.

Field Name	Description
`ItemID`	The item name.
`Value`	The data value.
`TimeStamp`	The time, as a MATLAB date vector, that the server's cache was updated.

Read async event information is stored in the EventLog property.

Example: `@opccallback`

Data Types: `char` | `cell` | `function_handle`

`RecordsAcquiredFcn` — Callback function to execute when `RecordsAcquired` event occurs
function handle | char | cell

You configure `RecordsAcquiredFcn` to execute a callback function file when a records acquired event is generated. A records acquired event is generated each time the RecordsAcquired property reaches a multiple of `RecordsAcquiredFcnCount`.

When a records acquired event occurs, the function specified in `RecordsAcquiredFcn` is passed two parameters: `Obj` and `EventInfo`. `Obj` is the object associated with the event, and `EventInfo` is an event structure containing the fields `Type` and `Data`. The `Type` field is set to `'RecordsAcquired'`. The `Data` field contains a structure with the fields defined below.

Field Name	Description
`LocalEventTime`	The time, as a MATLAB date vector, that the event occurred
`GroupName`	The group name
`RecordsAcquired`	The number of records acquired in the current logging session at the time the event occurred

Records acquired event information is not stored in the EventLog property.

Example: `@readNewRecords`

Data Types: `char` | `cell` | `function_handle`

`RecordsAcquiredFcnCount` — Number of records to acquire before `RecordsAcquired` event occurs
`20` (default) | positive integer

A records acquired event is generated each time the number of records acquired reaches a multiple of `RecordsAcquiredFcnCount`.

Example: `20`

Data Types: `double`

`StartFcn` — Callback function to execute immediately before logging starts
function handle | char | cell

You configure `StartFcn` to execute a callback function when all prelogging steps have been completed. You start logging by calling the `start` function. A start event occurs immediately before Logging is set to `'on'`.

When a start event occurs, the function specified in `StartFcn` is passed two parameters: `Obj` and `EventInfo`. `Obj` is the object associated with the event, and `EventInfo` is an event structure containing the fields `Type` and `Data`. The `Type` field is set to `'Start'`. The `Data` field contains a structure with the fields given below.

Field Name	Description
`LocalEventTime`	The time, as a MATLAB date vector, that the event occurred.
`GroupName`	The group name.
`RecordsAcquired`	The number of records acquired in the current logging session at the time the event occurred.

Start event information is stored in the `EventLog` property.

Example: `@opcLogStart`

Data Types: `char` | `cell` | `function_handle`

`StopFcn` — Callback function to execute immediately after logging stops
function handle | char | cell

You configure `StopFcn` to execute a callback function when logging has stopped. Logging stops when you issue a `stop` command, or when the RecordsAcquired value reaches RecordsToAcquire.

When a stop event occurs, the function specified in `StopFcn` is passed two parameters: `Obj` and `EventInfo`. `Obj` is the object associated with the event, and `EventInfo` is an event structure containing the fields `Type` and `Data`. The `Type` field is set to `'Stop'`. The `Data` field contains a structure with the fields given below.

Field Name	Description
`LocalEventTime`	The time, as a MATLAB date vector, that the event occurred.
`GroupName`	The group name.
`RecordsAcquired`	The number of records acquired in the current logging session at the time the event occurred.

Stop event information is stored in the EventLog property.

Example: @opcLogStop

Data Types: char | cell | function_handle

WriteAsyncFcn — Callback function to execute when asynchronous write completes
@opccallback (default) | function handle | char | cell

You configure WriteAsyncFcn to execute a callback function file when an asynchronous write operation completes. You execute an asynchronous write with the writeasync function. A write async event occurs immediately after the server notifies the client that data has written to the device.

When a write async event occurs, the function specified in WriteAsyncFcn is passed two parameters: Obj and EventInfo. Obj is the object associated with the event, and EventInfo is an event structure containing the fields Type and Data. The Type field is set to 'WriteAsync'. The Data field contains a structure with the fields defined below.

Field Name	Description
LocalEventTime	The time, as a MATLAB date vector, that the event occurred.
TransID	The transaction ID for the asynchronous write operation.
GroupName	The group name.
Items	A structure containing information about each item whose value or quality was written.

The Items structure contains the fields defined below.

Field Name	Description
ItemID	The item name.

Write async event information is stored in the EventLog property.

Example: @opccallback

Data Types: char | cell | function_handle

Subscription and Logging Settings

Active — Group activation state
'on' (default) | 'off'

Active can be 'on' or 'off'. If Active is 'on', the OPC server will return data for the group or item when requested by the read function or when the corresponding data items change (subscriptions). If Active is 'off', the OPC server will not return information about the group or item.

By default, Active is set to 'on' when you create the dagroup object. Set Active to 'off' when you are temporarily not interested in that daitem or dagroup object's values. You configure Active for both dagroup and daitem objects. Changing the state of the group does not change the state of the items.

The activation state of a dagroup or daitem object affects reads and subscriptions, and depends on whether the data is obtained from the cache or from the device. The active state of a group or item affects operations as follows.

Operation	Source	Active State
`read`	Cache	Both group and items must be active. Inactive items in active groups, and all items in inactive groups, return bad quality.
`read`	Device	Active is ignored.
`write`	N/A	Active is ignored.
`Subscription`	N/A	Both group and items must be active. Inactive items in active groups, and all items in inactive groups, return bad quality.
`readasync`	N/A	Active is ignored.

A transition from `'off'` to `'on'` results in a change in quality, and causes a subscription callback for the item or items affected. Changing the `Active` state from `'on'` to `'off'` will cause a change in quality but will not cause a callback since by definition callbacks do not occur for inactive items.

You enable subscription callbacks with the Subscription property. Use the DataChangeFcn property to specify a callback function file to execute when a data change event occurs.

Example: `'on'`

Data Types: `char`

LogFileName — Name of disk file to which logged data is written
`'opcdatalog.olf'` (default) | char

When you start a logging operation using the `start` function, and the LoggingMode property is set to `'disk'` or `'disk&memory'`, then `DataChange` events (records) are logged to a disk file with the name specified by `LogFileName`. You can specify any value for `LogFileName` as long as it conforms to the operating system file naming conventions. If no extension is specified as part of `LogFileName`, then `.olf` is used.

If a log file with the same name as `LogFileName` already exists when logging is started, the LogToDiskMode property is used to determine whether to overwrite the existing file, append records to that file, or create an indexed file based on `LogFileName`.

The log file is an ASCII file in comma-separated variable format, arranged as follows:

```
DataChange: LocalEventTime
ItemID1, Value1, Quality1, TimeStamp1
ItemID2, Value2, Quality2, TimeStamp2
...
ItemIDN, ValueN, QualityN, TimeStampN
DataChange: <LocalEventTime>
ItemID1, Value1, Quality1, TimeStamp1
ItemID2, Value2, Quality2, TimeStamp2
...
ItemIDN, ValueN, QualityN, TimeStampN
...
```

Example: `'opcdatalog.olf'`

Data Types: `char`

Logging — Status of data logging
`'off'` (default) | `'on'`

This property is read-only.

`Logging` is automatically set to `'on'` when you issue a `start` command. `Logging` is automatically set to `'off'` when you issue a `stop` command, or when the requested number of records is logged. You specify the number of records to log with the RecordsToAcquire property.

When `Logging` is `'on'`, each `DataChange` event (a record) is stored to disk or to memory (the buffer) as defined by the LoggingMode property.

Example: `'on'`

Data Types: `char`

LoggingMode — Specify destination for logged data
`'memory'` (default) | `'disk'` | `'disk&memory'`

`LoggingMode` can be set to `'disk'`, `'memory'`, or `'disk&memory'`, with the following effects:

- `'disk'` — `DataChange` events (records) are stored to the disk file specified by LogFileName.
- `'memory'` (default) — Records are stored to memory (the buffer).
- `'disk&memory'` — Records are stored to memory and to a disk file.

The disk file or memory buffer contains data logged from the time you issue the `start` command, until the time you issue a `stop` command or the number of records specified by the RecordsToAcquire property has been logged. Each `DataChange` event constitutes one record, containing one or more items. Only items that change value or quality are included in a `DataChange` event. The logged data includes the `ItemID`, `Value`, `TimeStamp`, and `Quality` for each item that changed.

Note that when you issue a `refresh` command while the toolbox is logging, the results of that operation are included in the log, since a `refresh` forces a `DataChange` event on the OPC server.

You extract data from memory with the `getdata` function. You can return the data stored in a log file to the MATLAB workspace with the `opcread` function.

Example: `'disk'`

Data Types: `char`

LogToDiskMode — Method of disk file handling for logged data
`'index'` (default) | `'append'` | `'overwrite'`

`LogToDiskMode` can be set to `'append'`, `'overwrite'`, or `'index'`, with the following effects:

- `'append'` — Data for a logged session is added to any data that already exists in the log file when logging is started using the `start` command.
- `'overwrite'` — The log file is overwritten each time `start` is called.
- `'index'` (default) — A different disk file is created each time `start` is called, according to the following rules:

  1. The first log file name attempted is specified by the initial value of `LogFileName`.
  2. If the attempted file name exists, then a numeric identifier is added to the value of `LogFileName`. For example, if `LogFileName` is initially specified as `'groupRlog.olf'`, then `groupRlog.olf` is the first attempted file, `groupRlog01.olf` is the second file name, and so on. If the `LogFileName` already contains numbers as the last characters in the file name,

then that number is incremented to create the new log file name. For example, if the `LogFileName` is specified as `'groupLog003.olf'`, then the next file name would be `'groupLog004.olf'`.

3. The actual file name used is the first file name that does not exist. In this way, each consecutive logging operation is written to a different file, and no previous data is lost.

Separate `dagroup` objects are logged to separate files. If two `dagroup` objects have the same value for `LogFileName`, then attempting to log data from both objects simultaneously results in the second object failing during the `start` operation.

Example: `'append'`

Data Types: `char`

`RecordsAcquired` — Number of records acquired
`0` (default) | double

This property is read-only.

`RecordsAcquired` is continuously updated to reflect the number of records acquired since the `start` function was called. When you issue a `start` command, the group object resets the value of `RecordsAcquired` to `0` and flushes the memory buffer.

To find out how many records are available in the buffer, use the RecordsAvailable property. You can also configure the RecordsAcquiredFcn to generate an event each time a particular number of records have been acquired.

Example: `20`

Data Types: `double`

`RecordsAvailable` — Number of records available in toolbox engine
`0` (default) | double

This property is read-only.

`RecordsAvailable` indicates the number of records that are available in the Industrial Communication Toolbox software engine. When you extract records from the engine with the `getdata` function, the `RecordsAvailable` value reduces by the number of records extracted. `RecordsAvailable` is reset to `0` and the toolbox engine is cleared when you issue a `start` command.

Use the RecordsAcquired property to find out how many records have been acquired since the `start` command was issued.

Example: `20`

Data Types: `double`

`RecordsToAcquire` — Number of records to acquire for logging session
`120` (default) | double

`RecordsToAcquire` specifies the number of records that must be acquired before the engine automatically stops logging. When RecordsAcquired reaches `RecordsToAcquire`, the Logging property is set to `'off'`, and no more records are logged.

To continuously log records, specify a value of `Inf`.

Example: `480`

Data Types: `double`

`Subscription` — Enable server update when data changes
`'on'` (default) | `'off'`

`Subscription` can be `'on'` or `'off'`. If `Subscription` is `'on'`, server update notification is enabled for the group. The update occurs when the server cache quality or value of the data associated with a `daitem` object contained by the `dagroup` object changes. In order for the server cache to be updated, the percent change in the item value must also be greater than the value specified for the DeadbandPercent property.

A `Subscription` value of `'on'` instructs the server to issue data change events when items in the group are updated by the server. Additionally, if an callback function file is specified for the DataChangeFcn property, that function executes. If `Subscription` is `'off'`, the server might still update item values or quality information, but no data change event is generated.

Note that the `refresh` function is a special case of subscription, where `refresh` forces a data change event for all active items.

Example: `'on'`

Data Types: `char`

Version History

Introduced before R2006a

See Also

Properties
opcda Object Properties Properties | daitem Object Properties Properties

Functions
`addgroup`

daitem Object Properties

Configure OPC `daitem` object

Description

Use the properties of the `daitem` object to examine item values, quality, timestamps, types, and so on.

Properties

General Settings

`AccessRights` — Inherent nature of access to item
`'read'` | `'read/write'` | `'write'`

This property is read-only.

`AccessRights` represents the server's ability to access a single OPC data item. The property value can be `'read'`, `'write'`, or `'read/write'`, with the following effects:

- `'read'` — You can read the server item's value.
- `'write'` — You can write values to the server item.
- `'read/write'` — You can read and change the server item's value. If you attempt a read or write operation on an item that does not have the required access rights, the server might return an error.

The value is set by the server when an item is created.

Example: `'read'`

Data Types: `char`

`CanonicalDataType` — Server's data type for item
char

This property is read-only.

`CanonicalDataType` indicates the data type of the item as stored on the OPC server. The MATLAB supported data types are indicated in the DataType property.

You can specify that the item's value is stored in the `daitem` object using a data type that differs from the canonical data type by setting the DataType property of the item to a value different from `CanonicalDataType`. Translation between the `CanonicalDataType` and the `DataType` is automatic.

Refer to the DataType property reference for a listing of the COM Variant data types and their equivalent MATLAB data types.

Data Types: `char`

DataType — OPC client item's data type

char

DataType indicates the data type of the item as stored in the daitem object in the MATLAB workspace. You can specify the data type when the item is created using the additem function. If you do not specify a data type, or if the requested data type is rejected by the server, the canonical (native) data type is used. If the client associated with the item is not connected, the data type is set to until the client is connected.

The OPC server uses this data type to store the item value. The CanonicalDataType property of a daitem object provides information on the canonical data type of that item on the server.

OPC communication uses COM Variant data types to send information between the server and client. These are automatically translated to an equivalent MATLAB data type for the COM Variant types defined below. Any data type not included in this list is returned as 'unknown'.

OPC Data Type	COM Data Type	MATLAB Data Type
double	VT_R8	double
char	VT_BSTR	char
single	VT_R4	single
uint8	VT_UI1	uint8
uint16	VT_UI2	uint16
uint32	VT_UI4	uint32
uint64	VT_UI8	uint64
int8	VT_I1	int8
int16	VT_I2	int16
int32	VT_I4	int32
int64	VT_I8	int64
currency	VT_CY	double
date	VT_DATE	double
logical	VT_BOOL	logical
double	VT_EMPTY	Empty array ([])

Example: 'double'

Data Types: char

ItemID — Fully qualified ID on OPC server

char

ItemID is the fully qualified ID of the data item on the OPC server. The server uses the ItemID to return the appropriate data from the server's cache, or to read and send data to a specific device or location.

You obtain valid ItemID values for a particular server by querying that server's name space using the getnamespace or serveritems functions.

Data Types: char

Parent — OPC object that contains this daitem object

OPC DA group object

This property is read-only.

For daitem objects, Parent indicates the dagroup object that contains the daitem object.

Data Types: DA group object

Quality — Quality of data value

Bad (default) | Good | Uncertain

This property is read-only.

Quality indicates the quality of the daitem object's Value property as a character vector. You can use the Quality property to determine if a value is useful or not.

The Quality is made up of a major quality, a substatus, and an optional limit status, arranged as a character vector in the format 'Major: Substatus: Limit status'. The limit status part is omitted if the value is not limited. The major quality can be one of the following values:

Value	Description
Bad	The value is not useful for reasons indicated by the substatus. The default value is 'Bad: Out of Service'.
Good	The value is of good quality.
Uncertain	The quality of the value is uncertain for reasons indicated by the Substatus.

For a list of substatus and limit status values and their interpretations, see "OPC Quality" on page A-2.

Quality is updated when you perform a read operation using read or readasync, or when a subscription callback occurs. Quality is also returned during a synchronous read operation.

Example: 'Bad: Out of Service'

Data Types: char

QualityID — Quality of data value as 16-bit integer

28 (default) | integer from 0 to 65535

This property is read-only.

QualityID is a numeric indication of the quality of the daitem object's data value.

QualityID is a number ranging from 0 to 65535, made up of four parts. The high 8 bits of the QualityID represent the vendor-specific quality information. The low 8 bits are arranged as QQSSSSLL, where QQ represents the major quality, SSSS represents the quality substatus, and LL represents the limit status.

You use the opcqparts function to extract the four quality fields from the QualityID value. Alternatively, you can use the bit-wise functions to extract the fields you are interested in. For example, to extract the major quality, you can bit-wise AND the QualityID with 192 (the decimal equivalent of binary `11000000`) using the `bitand` function, and shift the result 6 bits to the right using the `bitshift` function.

You use the `opcqstr` function to obtain the four quality fields from the `QualityID` value.

For more information, see "OPC Quality" on page A-2.

`QualityID` is updated when you perform a read operation using `read` or `readasync`, or when a subscription callback occurs.

Example: `28`

Data Types: `double`

`ScanRate` — Fastest possible data update rate
double

`ScanRate` describes the fastest possible rate at which a server can update an item. The default value is `0`, which indicates that the scan rate is not known. Note that the scan rate might not be attainable by the server due to network load, server load, and other factors.

The value is initially set by the server when a `daitem` object is created or when you connect to the server.

Data Types: `double`

Tag — Label to associate with OPC object
char

You configure `Tag` to be a character vector value that uniquely identifies an OPC object.

`Tag` is particularly useful when constructing programs that would otherwise need to define the toolbox object as a global variable, or pass the object as an argument between callback routines. You can return a toolbox object with the `opcfind` function by specifying the `Tag` property value.

Data Types: `char`

`TimeStamp` — Time when item was last read
date vector

This property is read-only.

`TimeStamp` indicates the time when the Value and Quality properties were obtained by the device (if this is available) or the time the server updated or validated `Value` and `Quality` in its cache. `TimeStamp` is updated when you perform an asynchronous or synchronous read operation or when a subscription callback occurs.

`TimeStamp` is stored as a MATLAB date vector. You can convert date vectors to date character vectors with the `datestr` function, and to MATLAB date numbers with the `datenum` function.

Data Types: `date vector`

Type — OPC object type
char

This property is read-only.

`Type` indicates the type of the object. The OPC object types are `'opcda'`, `'dagroup'`, and `'daitem'`. Once an object is created, the value of `Type` is automatically defined, and cannot be changed.

You can identify OPC objects of a given type using the `opcfind` function and the `Type` value.

Example: `'daitem'`

Data Types: `char`

`UserData` — Data to associate with OPC object

any type

You can configure `UserData` to store data that you want to associate with an OPC object. The object does not use this data directly, but you can access it using the `get` function.

`Value` — Item value

any MATLAB data type

This property is read-only.

`Value` indicates the value that was last obtained from the OPC server for the item defined by the ItemID property. The data type of the value is given by the DataType property.

The value returned from the server may be different from the value of the device to which the `ItemID` refers, if the DeadbandPercent for the `daitem` object's parent group is not zero. The value is also updated only periodically, based on the parent group's Active and UpdateRate properties.

You determine the validity of `Value` by checking the Quality property for the item.

`Value` is updated when you perform an asynchronous or synchronous read operation or when a subscription callback occurs.

Subscription and Logging Settings

`Active` — Item activation state

`'on'` (default) | `'off'`

`Active` can be `'on'` or `'off'`. If `Active` is `'on'`, the OPC server will return data for the group or item when requested by the `read` function or when the corresponding data items change (subscriptions). If `Active` is `'off'`, the OPC server will not return information about the group or item.

By default, `Active` is set to `'on'` when you create the `daitem` object. Set `Active` to `'off'` when you are temporarily not interested in that `daitem` object's values. You configure `Active` for both `dagroup` and `daitem` objects. Changing the state of the group does not change the state of the items.

The activation state of a `dagroup` or `daitem` object affects reads and subscriptions, and depends on whether the data is obtained from the cache or from the device. The active state of a group or item affects operations as follows.

Operation	Source	Active State
`read`	Cache	Both group and items must be active. Inactive items in active groups, and all items in inactive groups, return bad quality.
`read`	Device	Active is ignored.
`write`	N/A	Active is ignored.
`Subscription`	N/A	Both group and items must be active. Inactive items in active groups, and all items in inactive groups, return bad quality.
`readasync`	N/A	Active is ignored.

A transition from `'off'` to `'on'` results in a change in quality, and causes a subscription callback for the item or items affected. Changing the `Active` state from `'on'` to `'off'` will cause a change in quality but will not cause a callback since by definition callbacks do not occur for inactive items.

You enable subscription callbacks with the Subscription property. Use the DataChangeFcn property to specify a callback function file to execute when a data change event occurs.

Example: `'on'`

Data Types: `char`

Version History

Introduced before R2006a

See Also

Properties
opcda Object Properties Properties | dagroup Object Properies Properties

Functions
`additem`

Historical Data Access User's Guide

Introduction to OPC Historical Data Access (HDA)

- "OPC Historical Data Access" on page 12-2
- "Discover Available HDA Servers" on page 12-4
- "Connect to OPC HDA Servers" on page 12-5

OPC Historical Data Access

The OPC Historical Data Access (HDA) standard provides an interoperable platform to store and exchange historical process data. This standard differs from the OPC Data Access (DA) specification that deals only with real-time data. Industrial Communication Toolbox software provides a client interface to historical data access servers via the MATLAB environment. This client interface lets you:

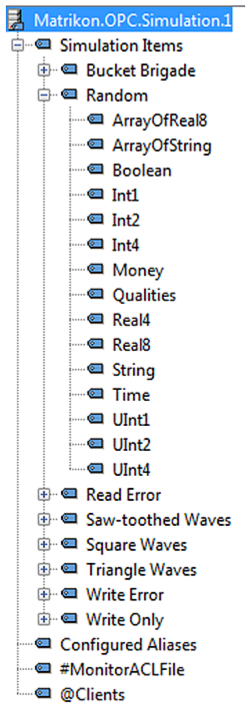
- Retrieve data from HDA servers into MATLAB
- Preprocess that data for common analysis tasks
- Visualize the data for easy interpretation

There are several types of OPC HDA historians:

- Simple trend data servers function only as basic raw data storage. The data itself would be of the type commonly made available by an OPC data access server and would take the form of value, quality, and timestamp triplets.
- Complex data compression and analysis servers provide data compression in addition to raw data storage. These servers are used where large volumes of process data are expected and storage space would be a limiting factor.
- Analysis servers are capable of providing analysis and summary information. They can support the updating of data and store the history of those updates. Storing data annotations may also be supported.

Industrial Communication Toolbox provides capabilities for reading raw and processed data from OPC HDA servers. Updating data on an HDA server and retrieving annotations is not supported.

Measurements from process end points (sensors, PLCs, etc.) are represented in the OPC HDA infrastructure as “items”. Each item has a unique item ID on the server, and therefore can be accessed uniquely. To best arrange the items, the server orders the items into a logical listing called a “name space.” These name spaces often take the form of a hierarchical tree in which groups of similar items are arranged into logical categories:



An item is usually represented by its fully qualified item ID (FQID) within the name space. An FQID is usually comprised of each level of the item's hierarchy separated by periods. For example:

```
Root.Branch1.Leaf3
```

In some cases, as in very small or simple historians, a hierarchical structure is not used. Instead all items are presented as a flat list of items.

Discover Available HDA Servers

In this section...
“Prerequisites” on page 12-4
“Determine HDA Server IDs for a Host” on page 12-4

Prerequisites

To interact with an OPC server, you must provide:

- The *host name* of the computer on which the OPC server is installed. Typically the host name is a descriptive term (such as `'plantserver'`) or an IP address (such as `192.168.2.205`).
- The *server ID* of the server you want to access on that host. Because a single computer can host multiple OPC servers, each server installed on that computer is given a unique ID during installation.

Your network administrator can provide the host names for all computers with OPC servers on your network. You can also obtain a list of server IDs for each host on your network, or use the `opcserverinfo` function to access server IDs from a host, as described next.

Determine HDA Server IDs for a Host

When an OPC server is installed, it must be assigned a unique server ID. This server ID provides a unique name for a particular instance of an OPC server on a host, even if multiple copies of the same server software are installed on that same machine.

To determine the server IDs of the OPC servers installed on a host, call the `opchdaserverinfo` function, specifying the host name as the only argument. When called with this syntax, the function returns a structure containing information about all the OPC servers available on that host:

```
info =
1x4 OPC HDA ServerInfo array:
 index    Host     ServerID                    HDASpecification           Description
 ----- ---------  --------------------------------  --------  ----------------------------------------------
  1    localhost  Advosol.HDA.Test.3                HDA1      Advosol HDA Test Server V3.0
  2    localhost  IntegrationObjects.OPCSimulator.1  HDA1      Integration Objects OPC DA DX HDA Simulator 2
  3    localhost  IntegrationObjects.OPCSimulator.1  HDA1      Integration Objects' OPC DA/HDA Server Simulator
  4    localhost  Matrikon.OPC.Simulation.1         HDA1      MatrikonOPC Server for Simulation and Testing
```

The fields in the structure returned by `opchdaserverinfo` provide this information:

Server Information Returned by opchdaserverinfo

Field	Description
`Host`	Character vector that identifies the name of the host. Note that no name resolution is performed on an IP address.
`ServerID`	Cell array containing the server IDs of all OPC servers accessible from that host.
`HDASpecification`	Cell array containing the OPC Specification that the server provides.
`Description`	Cell array containing descriptive text for each server.

Connect to OPC HDA Servers

Overview

After getting information about your OPC servers as described in “Discover Available HDA Servers” on page 12-4, you can establish a connection to the server by creating an OPC HDA client object, and connecting that client to the server. These steps are described next.

Note To run the sample code in the following steps you need the Matrikon OPC Simulation Server on your local machine. For installation details, see “Install an OPC DA or HDA Simulation Server for OPC Classic Examples” on page 1-14. The code requires only minor changes to work with other servers.

Create an HDA Client Object

Industrial Communication Toolbox does not use groups when dealing with HDA server items. Instead, the items themselves are passed to the available functions. These functions are accessible through the OPC HDA client object. In most cases, functions accessed via this HDA client object return an opc.hda.Data object. These data object simplify the display and manipulation of the historical data retrieved from the HDA server.

To create an OPC HDA client object, call the `opchda` function, specifying the host name and server ID. You retrieved this information using the `opchdaserverinfo` function (described in “Discover Available HDA Servers” on page 12-4). This example creates an OPC HDA client object to represent the connection to a Matrikon OPC Simulation Server:

```
hdaClient = opchda('localhost','Matrikon.OPC.Simulation.1');
```

View a Summary of a Client Object

To view a summary of the characteristics of the OPC HDA client object you created, enter the variable name you assigned to the object at the command prompt. For example, this is the summary for the `hdaClient` object:

```
hdaClient =
OPC HDA Client localhost/Matrikon.OPC.Simulation.1:
          Host: localhost
      ServerID: Matrikon.OPC.Simulation.1
       Timeout: 10 seconds
        Status: disconnected
    Aggregates: -- (client is disconnected)
ItemAttributes: -- (client is disconnected)
Methods
```

Connect an OPC HDA Client Object to the HDA Server

Use the `connect` function to connect a client to the server:

```
connect(hdaClient);
```

After connecting to the server, the Status information in the client summary display changes from `disconnected` to `connected`. If the client could not connect to the server (for example, if the OPC server is shut down), an error message appears. For information on troubleshooting connections to an OPC server, see “Troubleshooting OPC Issues” on page 1-17. After connecting to the client to the server, you can request a list of available aggregate types with the `hdaClient.Aggregates` function, as well as available item attributes with `hdaClient.ItemAttributes`. While connected you can browse the OPC server name space for information on available server items. See the next section for details on browsing the server name space. You can list the HDA functions with `methods(hdaClient)`.

Browse the OPC Server Name Space

A connected client object allows you to interact with the OPC server to obtain information about the name space of that server. The server name space provides access to all the data points provided by the OPC server by naming each data point with a server item, and then arranging those server items into a name space that provides a unique identifier for each server item.

The next section describes how to obtain a server name space or a partial server name space, using the `getnamespace` and `serveritems` functions.

Get an OPC HDA Server Name Space

Use the `getnamespace` function to retrieve the name space from an OPC HDA server. You must specify the client object that is connected to the server that you are interested in. The name space is returned as a structure array containing information about each node in the name space.

This example retrieves the name space of the Matrikon OPC Simulation Server installed on the local host:

```
hdaClient = opchda('localhost','Matrikon.OPC.Simulation.1');
connect(hdaClient);
ns = getnamespace(hdaClient)

ns =

3x1 struct array with fields:
    Name
    FullyQualifiedID
    NodeType
    Nodes
```

This table describes the fields of the structure:

Field	Description
`Name`	The name of the node, as a character vector.
`FullyQualifiedID`	The fully qualified item ID of the node, as a character vector. The fully qualified item ID is made up of the path to the node, concatenated with `'.'` characters. Use the fully qualified item ID when creating an item object associated with this node.
`NodeType`	The type of node. `NodeType` can be `'branch'` (contains other nodes) or `'leaf'` (contains no other branches).
`Nodes`	Child nodes. `Nodes` is a structure array with the same fields as `ns`, representing the nodes contained in this branch of the name space.

From the previous above, exploring the name space shows:

```
ns(1)
```

```
ans =
                Name: 'Simulation Items'
    FullyQualifiedID: 'Simulation Items'
            NodeType: 'branch'
               Nodes: [8x1 struct]
```

```
ns(3)

                Name: 'Clients'
    FullyQualifiedID: 'Clients'
            NodeType: 'leaf'
               Nodes: []
```

From this information, the first node is a branch node called `'Simulation Items'`. Since it is a branch node, it is most likely not a valid server item. The third node is a leaf node (containing no other nodes) with a fully qualified ID of `'Clients'`. Since this node is a leaf node, it is most likely a server item that can be monitored by creating an item object. To examine the nodes further down the tree, reference the `Nodes` field of a branch node. For example, the first node contained within the `'Simulation Items'` node is obtained as follows:

```
ns(1).Nodes(1)
```

```
ans =
                Name: 'Bucket Brigade'
    FullyQualifiedID: 'Bucket Brigade.'
            NodeType: 'branch'
               Nodes: [14x1 struct]
```

The returned result shows that the first node of `'Simulation Items'` is a branch node named `'Bucket Brigade'`, and contains 14 nodes.

```
ns(1).Nodes(1).Nodes(9)
```

```
ans =
                Name: 'Real8'
    FullyQualifiedID: 'Bucket Brigade.Real8'
            NodeType: 'leaf'
               Nodes: []
```

The ninth node in `'Bucket Brigade'` is named `'Real8'` and has a fully qualified ID of `'Bucket Brigade.Real8'`. Use the fully qualified ID to refer to that specific node in the server name space when creating items.

Using OPC HDA Client Objects

- “OPC HDA Objects” on page 13-2
- “Locate an OPC HDA Server” on page 13-3
- “Create an OPC HDA Client Object” on page 13-4
- “Connect to the OPC HDA Server” on page 13-5
- “Set Client Properties” on page 13-6
- “Retrieve an OPC HDA Server Name Space” on page 13-7
- “Read Item Attributes” on page 13-9

OPC HDA Objects

Industrial Communication Toolbox uses MATLAB objects to implement OPC HDA client functionality. The OPC HDA client object allows you to connect to the server and, when a connection is established, to access information about the server, retrieve the server's name space, and read data from the server. See “Create an OPC HDA Client Object” on page 13-4 for information on creating a client object.

By default, when data is read from the historian, the results are returned as OPC HDA data objects. These data objects provide a structured mechanism for storing OPC HDA data. Using data objects, you can visualize and manipulate historical data for later processing in MATLAB.

Before creating and connecting an OPC HDA client object to an OPC HDA server, you must locate the server on a particular host. The following sections describe how to locate, connect to, and browse the data on a server.

Locate an OPC HDA Server

To establish a connection between MATLAB and an OPC historical data access server, you obtain two pieces of information that the toolbox needs to uniquely identify the OPC historical data access server. You use this information when you create an OPC Historical Data Access (OPC HDA) client object.

The first piece of information is the host name of the server computer. The host name (a descriptive name like "HistorianServer" or an IP address such as 192.168.16.32) qualifies that computer on the network and is used by the OPC protocols to determine the available OPC servers on that computer. In any OPC application, you must know the name of the OPC server's host so that a connection with that host can be established. Your network administrator can provide a list of host names that provide OPC servers on your network. The following example uses `localhost` as the host name, because it connects to the OPC server on the same machine as the client.

The second piece of information is the OPC server ID. Each OPC server on a particular host is identified by a unique server ID (also called the Program ID or ProgID) allocated to that server on installation. The server ID is a character vector, usually containing periods. Although your network administrator can provide you with a list of server IDs for a particular host, you can query a host for all available OPC servers using the `opchdaserverinfo` function.

This example queries the local host for a list of available servers:

```
hostInfo = opchdaserverinfo('localhost')
```

```
hostInfo =
                 1x4 OPC HDA ServerInfo array:
    index    Host                ServerID              HDASpecification                  Description
    -----  ---------  --------------------------------  ----------------  ----------------------------------------------
      1    localhost  Advosol.HDA.Test.3                HDA1              Advosol HDA Test Server V3.0
      2    localhost  IntegrationObjects.OPCSimulator.1  HDA1              Integration Objects OPC DA DX HDA Simulator 2
      3    localhost  IntegrationObjects.OPCSimulator.1  HDA1              Integration Objects' OPC DA/HDA Server Simulator
      4    localhost  Matrikon.OPC.Simulation.1         HDA1              MatrikonOPC Server for Simulation and Testing
```

Examining the returned structure in more detail provides the server IDs of each OPC server:

```
allServers = {hostInfo.ServerID}
```

```
allServers =
Columns 1 through 3
    'Advosol.HDA.Test.3'    'IntegrationObjects.OPCSimulator.1'    'IntegrationObjects.OPCSimulator.1'
Column 4
    'Matrikon.OPC.Simulation.1'
```

Create an OPC HDA Client Object

After determining the host name and server ID of the OPC server you want to connect to, you can create an OPC HDA client object. The client controls the connection status to the server, stores properties of that server, and allows you to read data from the server.

Create an OPC HDA client using the `opchda` function, specifying the host name and server ID arguments:

```
hdaClient = opchda('localhost', 'Matrikon.OPC.Simulation.1')
```

```
hdaClient =
OPC HDA Client localhost/Matrikon.OPC.Simulation.1:
    Host: localhost
    ServerID: Matrikon.OPC.Simulation.1
    Timeout: 10 seconds

    Status: disconnected

    Aggregates: -- (client is disconnected)
    ItemAttributes: -- (client is disconnected)
```

You can also construct client objects directly from an OPC HDA `ServerInfo` object:

```
hostInfo = opchdaserverinfo('localhost');
hdaClient = opchda(hostInfo(1));
```

Connect to the OPC HDA Server

OPC HDA client objects are not automatically connected to the server when they are created. You can see this from the `Status` property of the client object.

Use the `connect` function to connect an OPC HDA client object to the server at the command line:

```
connect(hdaClient)
```

When connected, the client object properties update to show certain server properties:

```
hdaClient
```

```
hdaClient =
OPC HDA Client localhost/Matrikon.OPC.Simulation.1:
               Host: localhost
           ServerID: Matrikon.OPC.Simulation.1
            Timeout: 10 seconds

             Status: connected

         Aggregates: 6 Aggregate Types
     ItemAttributes: 10 Item Attributes
```

Browse the OPC Server Name Space

A connected client object allows you to interact with the OPC server to obtain information about the name space of that server. The server name space provides access to all the data points provided by the OPC server by naming each data point, and then arranging those server items into a name space that provides a unique identifier for each item. See “Retrieve an OPC HDA Server Name Space” on page 13-7.

Set Client Properties

You can modify many properties specific to the created client. These include `Timeout`, `UserData`, `Host` (before connection), and `ServerID` (before connection). Modify these properties as you would any other field of a MATLAB structure.

Set the Timeout Property

As OPC transactions often occur across networks, you might encounter cases where calls to those servers take some time to return. To change the function timeout of the OPC HDA client object, assign a new value to its `Timeout` property:

```
hdaClient.Timeout = 12
```

```
hdaClient =
OPC HDA Client localhost/Matrikon.OPC.Simulation.1:
               Host: localhost
           ServerID: Matrikon.OPC.Simulation.1
            Timeout: 12 seconds

             Status: connected

         Aggregates: 6 Aggregate Types
     ItemAttributes: 10 Item Attributes
Methods
```

Retrieve an OPC HDA Server Name Space

You use the `getNameSpace` function to retrieve the name space from an OPC HDA server. You must specify the client object that is connected to the server of interest. The name space is returned as a structure array containing information about each node in the name space.

This example retrieves the name space of the Matrikon OPC Simulation Server installed on the local host:

```
hdaClient = opchda('localhost','Matrikon.OPC.Simulation.1');
connect(hdaClient);
ns = getnamespace(hdaClient)
```

```
ns =
3x1 struct array with fields:
    Name
    FullyQualifiedID
    NodeType
    Nodes
```

This table describes the fields in the structure:

Field	Description
`Name`	The name of the node, as a character vector.
`FullyQualifiedID`	The fully qualified item ID of the node, as a character vector, often composed of the path to the node, concatenated with `'.'` characters. Use the fully qualified item ID when creating an item object associated with this node.
`NodeType`	The type of node. Can be `'branch'` (contains other nodes) or `'leaf'` (contains no other branches).
`Nodes`	Child nodes. Structure array with the same fields as `ns`, representing the nodes contained in this branch of the name space.

From the previous example, exploring the name space shows the following:

```
ns(1)
```

```
ans =
                Name: 'Simulation Items'
    FullyQualifiedID: 'Simulation Items'
            NodeType: 'branch'
               Nodes: [8x1 struct]
```

```
ns(3)
```

```
ans =
                Name: 'Clients'
    FullyQualifiedID: 'Clients'
            NodeType: 'leaf'
               Nodes: []
```

In this example, the first node is a branch node called `'Simulation Items'`. Because it is a branch node, it is probably not a valid server item. The third node is a leaf node (containing no other nodes) with a fully qualified ID of `'Clients'`. Because this node is a leaf node, it is most likely a server item that can be read. To examine the nodes further down the tree, you need to reference the `Nodes` field of a branch node. For example, the following code obtains the first node contained within the `'Simulation Items'` node:

```
ns(1).Nodes(1)
```

```
ans = 
                Name: 'Bucket Brigade'
    FullyQualifiedID: 'Bucket Brigade.'
            NodeType: 'branch'
               Nodes: [14x1 struct]
```

The result shows that the first node of `'Simulation Items'` is a branch node named `'Bucket Brigade'`, and contains 14 nodes.

```
ns(1).Nodes(1).Nodes(9)
```

```
ans = 
                Name: 'Real8'
    FullyQualifiedID: 'Bucket Brigade.Real8'
            NodeType: 'leaf'
               Nodes: []
```

The ninth node in `'Bucket Brigade'` is named `'Real8'` and has a fully qualified ID of `'Bucket Brigade.Real8'`. You use the fully qualified ID to refer to that specific node in the server name space when referencing items.

Read Item Attributes

Each item that you find on a server might have a given set of item attributes associated with it. These attributes provide information about the item stored on the server. The OPC Foundation defines a set of common item attributes, while specific servers can define server-specific attributes. However, support for item attributes is optional for any server.

You can find the attributes supported by your server by interrogating the `ItemAttributes` property of a connected HDA client object:

```
hdaClient.ItemAttributes
```

```
OPC HDA Item Attributes:
         Name           ID        Description
    --------------  ----------  ----------------
    DATA_TYPE           1       Data type
    DESCRIPTION         2       Item Description
    NORMAL_MAXIMUM      11      High EU
    NORMAL_MINIMUM      12      Low EU
    ITEMID              13      Item ID
    TRIANGLE        4294967291  Triangle Wave
    SQUARE          4294967292  Square Wave
    SAWTOOTH        4294967293  Saw-toothed Wave
    RANDOM          4294967294  Random
    BUCKET          4294967295  Bucket Brigade
```

You use the `readItemAttributes` function to retrieve the item attributes for a particular item.

For a list of OPC defined item attributes for the OPC HDA specification, see “OPC HDA Item Attributes” on page C-2.

Reading OPC Historical Data

- “Overview to Reading Historical Data” on page 14-2
- “Read Historical Data Over a Time Range” on page 14-3
- “Read Historical Data at Specific Times” on page 14-4
- “Read Processed Aggregate Data” on page 14-5
- “Retrieve Large Historical Data Sets” on page 14-6
- “Reading Modified Data” on page 14-7
- “Native MATLAB Data Types from Read Operations” on page 14-8
- “Disconnect from HDA Servers” on page 14-9
- “Clean Up OPC HDA Objects” on page 14-10

Overview to Reading Historical Data

After creating an OPC HDA client object (“Create an OPC HDA Client Object” on page 13-4) and connecting to the relevant server (“Connect to the OPC HDA Server” on page 13-5), you can access an array of functions which allow for the retrieval of historic data in various forms. The function you use depends on the type and range of data required as well as whether any aggregation or processing is required on that data.

The following table depicts the functions you can call to read certain types of data.

Function	Task or Condition
`readRaw`	Read data from the server as it was recorded, and process that data using MATLAB.
`readAtTime`	Read regularly sampled data or data from specific time stamps, and trust the interpolation algorithms used by the server.
`readProcessed`	The server processes data over a long time range, returning aggregates for particular intervals within that time range.
`readModified`	The server is capable of modifying data stored on the server, and you want to know what the values were before they were modified.

Read Historical Data Over a Time Range

The `readRaw` function allows you to request the value, quality, and timestamp data for a list of items over a specified time domain. Define the time domain by indicating start and end times for the sampling. This function returns all data stored on the historian within the given time range.

By default, historians return the first data point found from the start time specified, up to the data point found just before the end time. By setting the optional `'bounds'` parameter to `true`, you can indicate that bounding values be included. The server then returns data at the start and end times. If no data exists at those exact times, the server returns the data value that is closest to that time but outside the time range specified.

This function is useful if you want to retrieve raw values from the server, and processes that data using MATLAB rather than relying on the server to perform the processing for you.

For example, if you are interested in the values between 17 November 2010 and 18 November 2010 in the `'Int2'` items under the `'Random'` branch of an OPC HDA server, and you were interested in retrieving the bounding values, use this code:

```
DataObject = ReadRaw(HdaClient, 'Random.Int2', ...
       datenum(2010,11,17), datenum(2010,11,18), TRUE)
```

To read values at specified time stamps use the `readAtTime` function. If you are reading large amounts of data and will be aggregating that data, consider using `readProcessed` (if your server supports that function).

Read Historical Data at Specific Times

The `readAtTime` function reads the values for a list of item IDs at specific times. This is useful if your analysis routine requires regularly sampled data and you can accept the interpolation scheme used by your server. If no value exists on the server at the exact timestamp requested, the value is interpolated from the surrounding data values.

For example, if you wanted the values of two items at this current moment and their values at the same time yesterday, you could use the following code:

```
itemList = {'Random.Int1', 'Random.Boolean'}
timeStamps = [now; now-1];
dataObject = readAtTime(hdaClient, itemList, timeStamps)
```

Additionally, you can request that the data be returned as a supported MATLAB data type. See “Native MATLAB Data Types from Read Operations” on page 14-8.

The same example could be called, but with a MATLAB data type specified as a fourth parameter. This function call returns all the data values as 8-bit signed integers:

```
dataObject = readAtTime(HdaClient, ItemList, TimeStamps, 'int8')
```

You can now use this object as required, or display it as described in “Display Data Objects” on page 15-3.

Read Processed Aggregate Data

Historians can include the ability to process raw data in a variety of ways before returning it to you. Examples of such processing include the interpolation of data points, time averaging, and standard deviation calculations. Processing of data can be very useful when there is a large amount of data on the server. Instructing the server to return only a processed data set can greatly reduce the time and volume of data transferred.

You can discover which aggregates are supported by the server by requesting the `Aggregates` property of a connected HDA client object:

```
aggTypes = clientObject.Aggregates
aggTypes =
OPC HDA Aggregate Types:
          Name          ID                                            Description
    -----------------  --  --------------------------------------------------------------------------------------------
    INTERPOLATIVE       1   Retrieve interpolated values.
    TIMEAVERAGE         4   Retrieve the time weighted average data over the resample interval.
    MINIMUMACTUALTIME   7   Retrieve the minimum value in the resample interval and the timestamp of the minimum value.
    MINIMUM             8   Retrieve the minimum value in the resample interval.
    MAXIMUMACTUALTIME   9   Retrieve the maximum value in the resample interval and the timestamp of the maximum value.
    MAXIMUM            10   Retrieve the maximum value in the resample interval.
```

In the previous example, the server supports six types of aggregate.

You can request processed data using the `readProcessed` function and passing in the ID of the aggregate required. You can retrieve the property ID using the object and the appropriate aggregate type.

```
clientObject.Aggregates.TIMEAVERAGE

     4

hdareadProcessed = readProcessed(clientObject, ItemList, clientObject.Aggregates.TIMEAVERAGE, ...
                      AggregateInterval, StartTime, EndTime)
hdareadProcessed =
1-by-5 OPC HDA Data object:
       ItemID         Value           Start TimeStamp           End TimeStamp                Quality
    ------------  --------------  -----------------------  -----------------------  -----------------------------
    Random.Int1    1 int8 value     2010-11-28 13:56:40.666  2010-11-29 13:56:40.666  1 unique quality [Calculated]
    Random.Boolean 1 logical value  2010-11-28 13:56:40.666  2010-11-29 13:56:40.666  1 unique quality [Calculated]
```

The requested time domain is split into the time intervals you provide as the fourth function argument. The aggregates are calculated over these intervals.

Additionally, you can request that the data be returned as a supported MATLAB data type. See “Native MATLAB Data Types from Read Operations” on page 14-8.

Retrieve Large Historical Data Sets

This example shows how to retrieve very large data sets from OPC historical data access servers.

Your OPC HDA server may have a defined upper limit on how much data to return in any given historical data access read operation. That upper limit is returned by the `MaxReturnValues` field of the structure returned by calling `getServerStatus` on the client object. A value of 0 means there is no defined limit, and the server returns all possible values.

When you request data over a wide time range, the server returns up to `MaxReturnValues` elements for each item, and the read function issues a warning. The warning ID is `opc:hda:mex:ReadMoreData`. To retrieve all values, use code similar to that shown here.

This example retrieves all values of two items over a full year.

```
lastwarn('');
startTime = datenum(2013,1,1);  % Replace with your start time
endTIme = datenum(2013,12,31);  % Replace with your end time
itmList = {'Plant1.Unit2.FIC1001', 'Plant2.Unit1.FIC1001'}; % Replace with your item list
wState = warning('off','opc:hda:mex:ReadMoreData');
yearData = hdaObj.readRaw(itmList,startTime,endTime);
[warnMsg, warnID] = lastwarn;
gotAllData = isempty(strfind(warnID,':ReadMoreData')) && isempty(strfind(warnID,':ReadComposite'));
while ~gotAllData
    % Update start time to last time retrieved
    endDates = cellfun(@(x)x(end), {yearData.TimeStamp});
    startTime = max(endDates);
    % Read data and append to existing data set
    moreData = hdaObj.readRaw(itmList,startTime,endTime);
    yearData = append(yearData,moreData);
    [warnMsg, warnID] = lastwarn;
    gotAllData = isempty(strfind(warnID,':ReadMoreData'));
end
% Reset warning state
warning(wState);
```

Reading Modified Data

It is possible that at some point historical data might be modified on the server, and you are interested in these changes. In this case you would use `readModified` function. This function returns the timestamps at which the data was modified and the value before that modification. If `readRaw`, `readAtTime`, or `readProcessed` returns a quality value of `OPCHDA_EXTRADATA`, it indicates that the item in question has been modified and more information can be retrieved using `readModified`. By providing the function with a list of items that you are interested in and the time range over which you would like to query for changes, you can retrieve any changed data items. This function operates similarly to `readRaw`, but only modified data is returned.

Native MATLAB Data Types from Read Operations

The default format of returned data is an `M`-by-1 OPC HDA data object containing data values whose type is defined by the OPC variant type the server stored it as. In some cases, such as `readAtTime` and `readProcessed`, you can specify that the read operations return data in native MATLAB data types, including structures and cell arrays.

For example, you can request the same set of data in the following ways.

Request Structure Output

In this case, the read operation returns a single output containing four fields:

```
struct =  HDAObject.readAtTime('Random.Int1', TimeStamps, 'struct')
struct =
       ItemID: 'Random.Int1'
    Timestamp: [8x1 double]
      Quality: [8x1 double]
        Value: [8x1 int8]
```

Request MATLAB Numeric Data Output

When you request MATLAB numeric types as output, the read operation returns four outputs: Item ID, Value, Quality, and TimeStamp. The Value output is converted into the MATLAB data type requested. The following example returns all Value data as unsigned 32-bit integers:

```
[itmId, val, Q, ts] = HDAObject.readAtTime('Random.Int1', TimeStamps, 'uint32');
```

Request Cell Array Output

When requesting cell array output, the read operation returns four outputs: Item ID, Value, Quality, and TimeStamp. The Value output is a cell array, preserving the original data type of the item on the server.

```
[cItemId, cVal, cQ, cTimes] = HDAObject.readAtTime('Random.Int1', TimeStamps, 'cell')
```

Disconnect from HDA Servers

Disconnecting a client releases the client object from the server and frees system resources. Do this by calling the `disconnect` command on the client object:

```
disconnect(hdaObject)
```

Clean Up OPC HDA Objects

Disconnecting a client does not delete the client object from the MATLAB workspace, nor does it remove any data objects created during reads executed via the client object. You can remove these objects from the workspace using the MATLAB `clear` command:

```
clear hdaObj
clear dataObj
```

Working with OPC HDA Data Objects

- “Introduction to OPC HDA Data Objects” on page 15-2
- “Display Data Objects” on page 15-3
- “OPC HDA Quality Values” on page 15-4
- “Manipulate Data Using OPC HDA Objects” on page 15-5

Introduction to OPC HDA Data Objects

All data returned from OPC HDA servers can be stored in MATLAB as an OPC HDA data object. The HDA data object allows for convenient data storage, manipulation, and visualization. The data elements themselves are represented by one or more value, quality, and timestamp values, all associated with an item ID.

When you perform read operations on OPC HDA servers, you request data for one or more item IDs on that server over a specified time range. For each item requested, the OPC server returns zero or more data object elements stored as triplets of Value (the sensor reading or item value), Quality (the quality of the value stored), and TimeStamp (the time the data was logged by the server). The Value, Quality, and TimeStamp properties are always `M`-by-1 vectors. The data type of the Value property depends on what the server returns to MATLAB. See “Conversion Between MATLAB Data Types and COM Variant Data Types” on page 8-13.

Each read operation thus returns an array of OPC HDA data objects, one for each item requested. Elements of a data object array are not guaranteed to have the same number of Value, Quality, and TimeStamp triples, because the server might not have logged data at the same time for all items requested.

Display Data Objects

OPC HDA data read operations can produce a large amount of data returned to MATLAB. To accommodate this, Industrial Communication Toolbox provides two functions to display data objects. By default, a summary of the data is presented. To display data in this form, type the object name at the MATLAB command line, similar to this:

```
myDataObject
```

```
1-by-1 OPC HDA Data object:
       ItemID          Value           Start TimeStamp           End TimeStamp                Quality
    ------------  ---------------  -----------------------  -----------------------  -----------------------------
    Scalar.Item1  8 double values  2010-10-13 14:18:11.832  2010-11-11 14:18:11.832  1 unique quality [Extra Data]
```

The `showValues` function displays the internal values of the data object in a table. This form is preferable if you want all the data values to be visible, for example when generating reports or visually scanning the data.

```
myDataObject.showValues
```

```
OPC HDA Data object for item Scalar.Item1:
           TIMESTAMP              VALUE          QUALITY
    =======================  =============  ================
    2010-10-13 14:18:11.832       3.000000  Extra Data (Bad)
    2010-10-18 14:18:11.832      37.000000  Extra Data (Bad)
    2010-10-22 14:18:11.832      17.000000  Extra Data (Bad)
    2010-10-23 14:18:11.832      21.000000  Extra Data (Bad)
    2010-11-01 14:18:11.832      25.000000  Extra Data (Bad)
    2010-11-09 14:18:11.832      38.000000  Extra Data (Bad)
    2010-11-10 14:18:11.832      31.000000  Extra Data (Bad)
    2010-11-11 14:18:11.832      39.000000  Extra Data (Bad)
```

OPC HDA Quality Values

OPC HDA quality values identify the quality or integrity of retrieved historical data. The quality is returned as a 32-bit number with only the upper 16 bits relating specifically to HDA; the lower 16 bits relate to both OPC data access. For information on data access quality, see “OPC Quality” on page A-2.

Upper 16-bit HDA Quality Values

Quality Values	Description	Mask Value	Associated DA Quality
`OPCHDA_EXTRADATA`	More than one piece of data that might be hidden exists at same timestamp.	0x00010000	Good, Bad, Quest
`OPCHDA_INTERPOLATED`	Interpolated data value.	0x00020000	Good, Bad, Quest
`OPCHDA_RAW`	Raw data value.	0x00040000	Good, Bad, Quest
`OPCHDA_CALCULATED`	Calculated data value, as would be returned from a ReadProcessed call.	0x00080000	Good, Bad, Quest
`OPCHDA_NOBOUND`	No data found to provide upper or lower bound value.	0x00100000	Bad
`OPCHDA_NODATA`	No data collected. Archiving not active (for item or all items).	0x00200000	Bad
`OPCHDA_DATALOST`	Collection started / stopped / lost.	0x00400000	Bad
`OPCHDA_CONVERSION`	Scaling / conversion error.	0x00800000	Bad, Quest
`OPCHDA_PARTIAL`	Aggregate value is for an incomplete interval.	0x01000000	Good, Bad, Quest

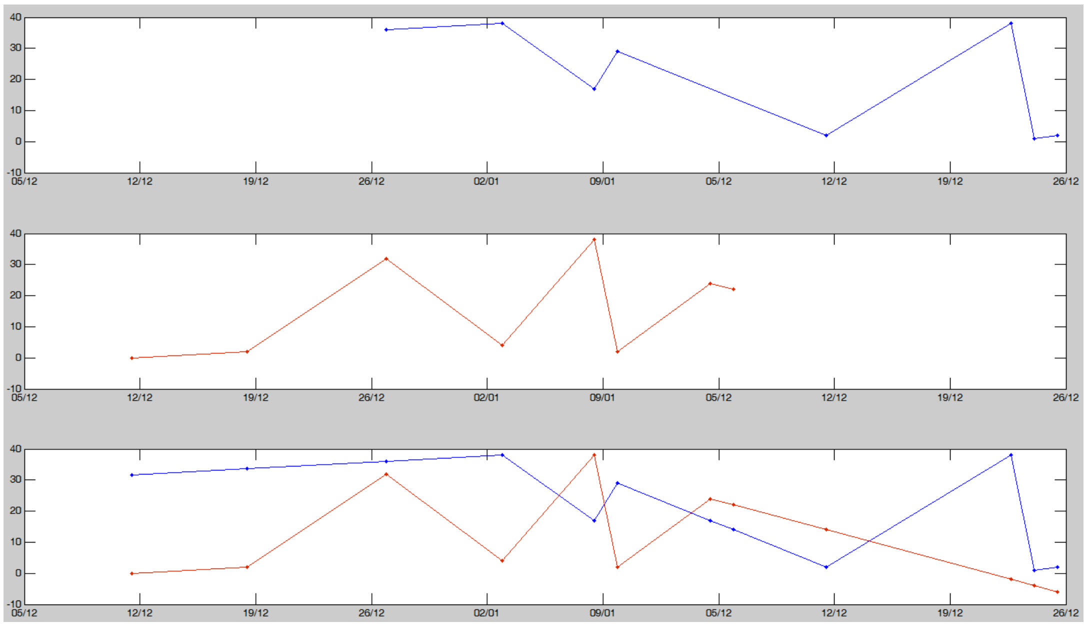
Manipulate Data Using OPC HDA Objects

OPC HDA data objects provide initial data storage, visualization, and manipulation functions for you to work with OPC historical data in MATLAB. To facilitate preparation for further processing, OPC HDA data objects allow you to resample OPC historical data as follows:

- To prepare data for analysis algorithms that require data to be regularly sampled, use the `resample` function.
- To ensure that data from all items contains the same timestamp vector, use the `tsunion` function, which keeps all data and interpolates data for missing timestamps in each item, or the `tsintersect` function, which discards any data from a timestamp that does not exist in all items in the object.

Resample Data Objects to Include All Available Time Stamps Using tsunion

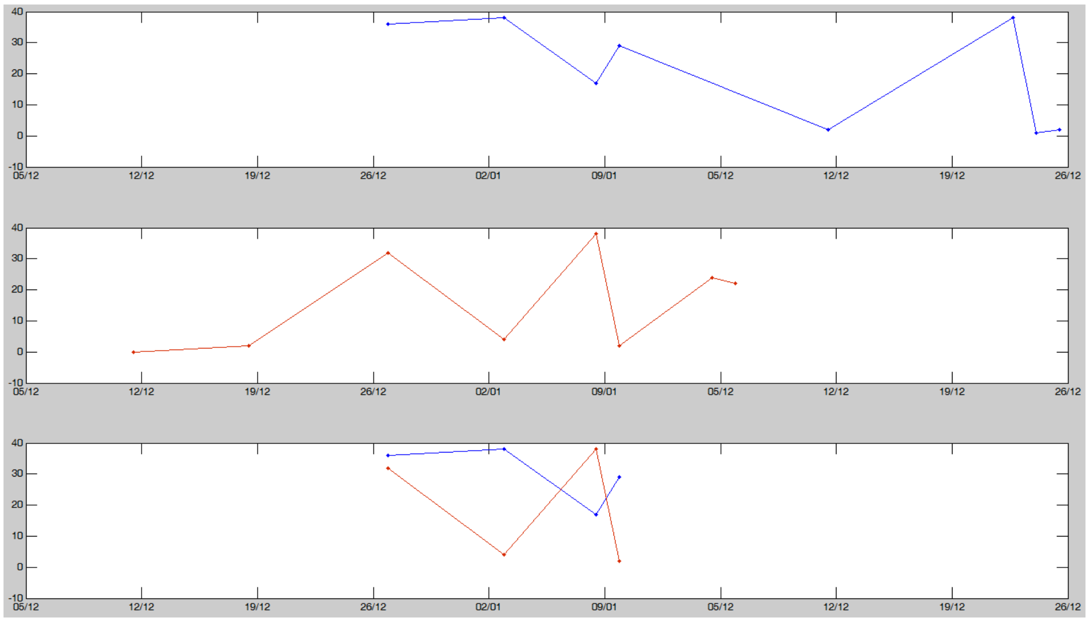
Given an array of data objects, `tsunion` adapts all data to have a single common set of timestamps by finding all unique time stamps in all items of the array. The values of each data item are then extrapolated or interpolated at the new timestamps. Resampling is performed using the method specified in the function call. Valid methods are `'linear'`, `'spline'`, `'pchip'`, `'nearest'`, and `'hold'`. The default is `'linear'`. If any returned `Value` is a character vector, only `'hold'` is supported. Elements with the same item ID are combined, so that `tsunion` creates data objects with unique item IDs. The `Quality` of interpolated timestamps is set to `'Interpolated:Good'`, and for extrapolated timestamps is set to `'Interpolated:Uncertain'`.



The top two plots above depict two separate data objects. The bottom plot is the result of these two data objects being passed to the `tsunion` function. You can see that in the bottom plot that each element has been extended to include the timestamps of the other and that values have been extrapolated to satisfy these new timestamps.

Resample Data Objects to Include All Common Time Stamps Using tsintersect

When you are interested in only the timestamps common to a number of data objects, you can use the `tsintersect` function. It generates a new OPC HDA data object in which each element has the same timestamp vector composed of those timestamps that were common to all items in the original data objects provided. If the provided data objects contain elements with the same item ID, those elements are combined into one before computing the intersection.



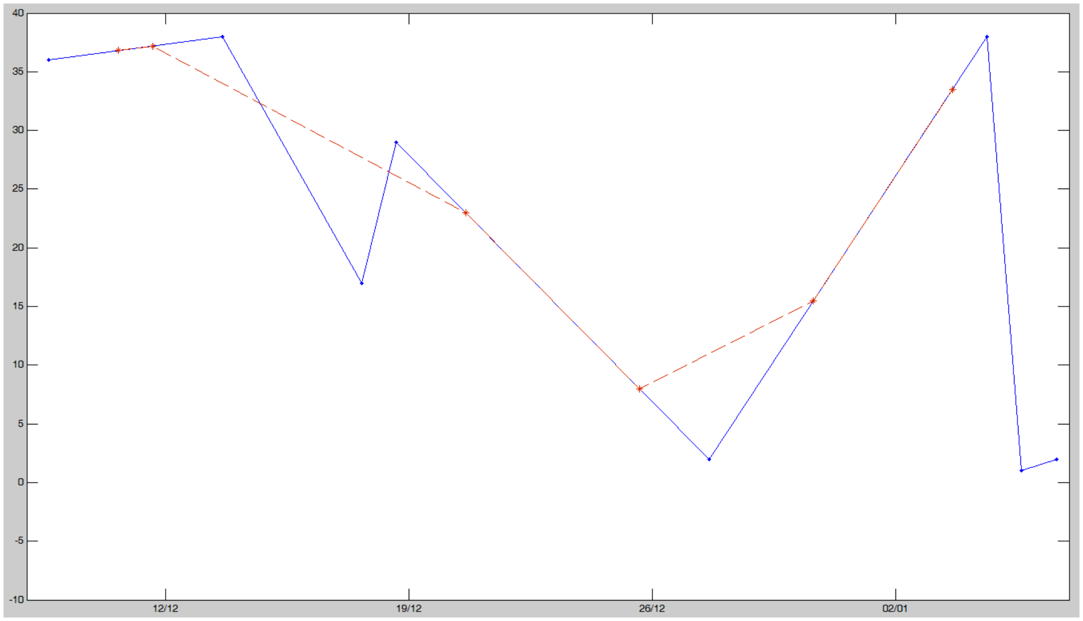
The previous figure shows how the values of two data objects, plotted in the first and second positions respectively, can be intersected to produce a new object whose elements contain only timestamps common to the original two. Uncommon timestamps are discarded along with their data values.

Resample Data to a New Set of Time Stamps

You might want to resample all items in a data object at specified time stamps; for example, when you have data values for a second item and want to correlate your data object with the original at the same timestamps. Where no exact values are available, the `resample` function resamples (interpolate or extrapolate) the data values at the requested time stamps using the resampling method you specify. Valid methods include `'linear'`, `'spline'`, `'pchip'`, and `'nearest'` (see `interp1` for details on these methods), as well as `'hold'`, which implements a zero-order-hold behavior (previous values are held until a new value exists).

For character vector values, only the `'hold'` method is supported. Trying to resample data containing character vectors with any method other than `'hold'` generates an error.

This concept is illustrated in the following graphic.



In this figure, the blue line represents the original data values while the red line represents the resampled data at a new set of timestamps. These new timestamps are marked by red stars while the original timestamps are marked by blue circles.

Convert OPC HDA Data Objects to MATLAB Numeric Data Types

When retrieving data from the server and storing it in an OPC data object, the client automatically converts the values from the OPC variant types (see Comparison of MATLAB and COM Variant Data Types). Retrieve the data values from the data object by referencing the `Value` property. For example, to display and access the first element of the `hdaReadRaw` data object:

```
hdaReadRaw
```

```
hdaReadRaw =
1-by-5 OPC HDA Data object:
        ItemID            Value            Start TimeStamp           End TimeStamp            Quality
    --------------   -----------------   -----------------------   -----------------------   ----------------------
    Random.Int1       5 int8 values    2010-12-01 16:05:30.902  2010-12-01 16:05:32.869  1 unique quality [Raw]
    Random.Uint2     5 double values   2010-12-01 16:05:30.902  2010-12-01 16:05:32.869  1 unique quality [Raw]
    Random.Real8     5 double values   2010-12-01 16:05:30.902  2010-12-01 16:05:32.869  1 unique quality [Raw]
    Random.String     5 cell values    2010-12-01 16:05:30.902  2010-12-01 16:05:32.869  1 unique quality [Raw]
    Random.Boolean   5 logical values  2010-12-01 16:05:30.902  2010-12-01 16:05:32.869  1 unique quality [Raw]

class(hdaReadRaw(1).Value)

int8
```

An alternative is to call standard type conversion methods available in MATLAB on the entire object, in which case all items are converted to the chosen type (assuming they have the same timestamp vectors):

```
newArray = double(hdaReadRaw(1));
class(newArray)

double
```

In this example, `hdaReadRaw(1)` has an initial native data type of `'int8'`, yet after passing it to the `'double'` conversion call, the resulting values are of the native MATLAB type `'double'`.

OPC HDA and UA Classes

opc.hda.AggregateTypes class

Package: `opc.hda`

OPC HDA server aggregate types

Construction

You do not create `AggregateTypes` objects directly; instead, when you connect an OPC HDA client to the server, the `Aggregates` property is automatically populated with available aggregate types for that server.

Methods

getDescription	Get description of OPC HDA aggregate type or item attribute
getIDFromName	Translate OPC HDA aggregate type or item attribute name to numeric identifier
getIDList	Get all aggregate type or item attribute IDs
getNameList	Get all aggregate type or item attribute names

Properties

`AggregateTypes` objects have no generic user-visible properties. Instead, each available aggregate type is created as a property. For example, if the server supports the `TIMEAVERAGE` aggregate type, the `AggregateTypes` object stored in the `Aggregates` property of a client connected to that server has a property named `TIMEAVERAGE` with its value set to the numeric ID of that attribute.

Copy Semantics

Value — To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also

`opc.hda.Client` | `readProcessed`

opc.hda.Data class

Package: `opc.hda`

OPC HDA data object

Description

The `opc.hda.Data` object stores and presents information retrieved from an OPC historical data access server. The OPC HDA data object allows you to store and process data retrieved from an OPC HDA server, and convert that data into MATLAB data types that can be operated on further.

Construction

You construct OPC HDA data objects using the various methods to read an OPC HDA client object.

Methods

Properties

Copy Semantics

Value — To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also

`readRaw` | `readAtTime` | `readProcessed` | `readModified`

opc.hda.ItemAttributes class

Package: `opc.hda`

OPC HDA item attributes

Description

OPC servers store and publish item attributes for each item in the server's name space. Such attributes assist in describing items, including their scaling, limits, and data types. A server is not obliged to store attributes, although common attributes are defined in the OPC HDA specification.

The `ItemAttributes` class is used to store item attributes available on a server. You do not create `ItemAttributes` objects directly; instead, when you connect an OPC HDA client to the server, the `ItemAttributes` property is automatically populated with available item attributes for that server.

You can access the required aggregate type using dot-notation on the `ItemAttributes` property of a connected OPC HDA client. For example, for client `hdaObj`, you can access the `MAXIMUM` attribute by typing `hdaObj.ItemAttributes.MAXIMUM`. Tab completion works for item attributes. Specific attributes are distinguished from class methods by all-capitals: `getDescription` is not an available aggregate type, but is a method of the `ItemAttributes` class.

Construction

You do not create `ItemAttributes` objects directly; instead, when you connect an OPC HDA client to the server, the `ItemAttributes` property is automatically populated with available item attributes for that server.

Methods

getDescription	Get description of OPC HDA aggregate type or item attribute
getIDFromName	Translate OPC HDA aggregate type or item attribute name to numeric identifier
getIDList	Get all aggregate type or item attribute IDs
getNameList	Get all aggregate type or item attribute names

Properties

`ItemAttributes` objects have no generic user-visible properties. Instead, each available item attribute is created as a property. For example, if the server supports the `DESCRIPTION` item attribute, the `ItemAttributes` object stored in the `ServerItemAttributes` property of a client connected to that server has a property named `DESCRIPTION` with the value set to the numeric ID of that attribute.

Copy Semantics

Value — To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also

`opc.hda.Client` | `readItemAttributes`

opc.hda.ServerInfo class

Package: `opc.hda`

OPC HDA server information objects

Description

The `ServerInfo` class stores information about installed OPC HDA servers on a specified host. You can use `ServerInfo` objects to quickly construct OPC HDA clients associated with a particular OPC HDA server.

Construction

You should not directly create this class. Instead, use `opchdaserverinfo` to retrieve information about servers from a particular host.

Methods

findDescription	Locate OPC HDA servers with particular description

Properties

Copy Semantics

Value — To learn how this affects your use of the class, see Comparing Handle and Value Classes in the MATLAB Object-Oriented Programming documentation.

See Also

`opchdaserverinfo`

Unified Architecture User’s Guide

OPC Unified Architecture (UA)

- “About OPC Unified Architecture” on page 17-2
- “OPC UA Components” on page 17-3
- “OPC UA Server Data Types” on page 17-5
- “OPC UA Security” on page 17-7
- “OPC UA Certificate Management” on page 17-9
- “OPC UA Aggregate Functions” on page 17-10
- “Access Data from OPC UA Servers” on page 17-13

About OPC Unified Architecture

The OPC Unified Architecture (OPC UA) standard combines all the capabilities of OPC Data Access and OPC Historical Data Access standards (together, referred to as "OPC Classic") and adds various additional capabilities into a single, extensible standard. OPC UA servers provide a single namespace which organizes the data available on the server into a hierarchical view of nodes (also called items in OPC Classic terminology). Nodes on OPC UA servers can be object nodes, which organize other nodes, or variable nodes, which have a value representing some process value on the server. Variable nodes can contain other variable nodes. Nodes are arranged in a number of representations; for Industrial Communication Toolbox, the nodes are exposed as a hierarchical tree, with nodes containing subnodes called children.

OPC UA servers are required to publish a node in their namespace named "Server". The Server node provides information about the OPC UA server, including the capabilities of the server, specific limitations of the server, and other information related to the server. Industrial Communication Toolbox provides selected information from the Server node as properties of the client you create to connect to that server. For information on client properties, see `opc.ua.Client`.

OPC UA servers may or may not historize Variable nodes. For historizing nodes, OPC UA servers store prior values of the node, and can provide that history to OPC UA Clients as raw data (data points at the times that the server stored the Value), or as data at requested times (the server interpolates the raw data using either sample-and-hold or linear interpolation), or as processed data, using a predefined aggregate function that is requested by the user. Each OPC UA server describes which Aggregate Functions are supported by that server. "OPC UA Aggregate Functions" describes the standard aggregate functions defined in the OPC UA specification. Servers may implement custom aggregate functions; consult the specific OPC UA server reference for information on how those functions work. Industrial Communication Toolbox provides a client interface to the OPC UA servers which allows you to browse the server namespace to find nodes of interest. The toolbox supports the opc.tcp binary protocol and anonymous, unsecured connections. You can also use the client to define the security configuration for the connection, and provide user credentials to the server. The toolbox supports the opc.tcp binary protocol only; HTTP and HTTPS connections are not supported. For some of the tasks you can perform with Industrial Communication Toolbox, see the related examples.

See Also

Related Examples

- “Read and Write Current OPC UA Server Data” on page 21-51
- “Read Historical OPC UA Server Data” on page 21-56
- “Visualize and Preprocess OPC UA Data” on page 21-61

OPC UA Components

In this section...

“Overview” on page 17-3
“OPC UA Client” on page 17-3
“OPC UA Node” on page 17-3
“OPC UA Data” on page 17-4
“OPC UA Quality” on page 17-4
“Working with Time in OPC UA” on page 17-4

Overview

Industrial Communication Toolbox provides an OPC UA client to connect to OPC UA servers. Using the client, you connect to the server, query server status, browse the server namespace, read and write current values, and read historical values from nodes on the server. Historical data is retrieved as OPC data objects, which allow you to process historical data in preparation for common analysis tasks.

OPC UA Client

You construct the OPC UA client using the `opcua` function. You set the security configuration for the connection using `setSecurityModel`. You connect the client to the server using `connect`, optionally passing user authentication credentials. The client includes a number of properties describing the server capabilities, including supported security models and user authentication options. See `opc.ua.Client` for more information on the properties available to the client. You can also query the server for extended status information using `getServerStatus`.

You use the client to perform any communication with the server, including browsing the server name space, reading and writing current values, and reading historical values from the server.

OPC UA Node

The OPC UA client includes a `Namespace` property, which contains the top level of the server’s namespace as an array of Nodes. An OPC UA Node variable describes the node on the server, and contain other subnodes in the `Children` property. Nodes have a `NodeType` which can be `'Object'` or `'Variable'`. Object nodes have no value associated with them, and are used purely for organizing the namespace of the server. Variable nodes store current values, representing a sensor or actuator value associated with the server. For more information, see `opc.ua.Node`

Servers can choose to historize nodes (store previous data values for that node). The `Historizing` property of a Node defines whether a server is historizing the node or not. If you try to retrieve historical data from a Variable node with `Historizing` set to `false`, no data is returned and an error is displayed.

You can read and write current values, and retrieve historical data, using Node variables directly. This is simply a short-hand for performing the same operations on the node `Client` property.

OPC UA Data

Data retrieved from OPC UA servers includes three important values. The Value is accompanied by a Quality and a Timestamp. The Quality represents how accurately the data Value is considered to reflect the actual source value attached to the server. The Timestamp represents the time that the server recorded the value, or received notification from the data source that the value is current.

When you retrieve current values, the Value, Quality, and Timestamp are retrieved into separate arrays. When you retrieve historical values, OPC UA servers might return a different number of Value, Quality, and Timestamp arrays for each Node requested. This data is packaged into an OPC UA Data object, which allows you to process this data set in preparation for common analysis tasks. For more information, type

```
help opc.ua.Data
```

For an example of working with OPC UA data, see “Visualize and Preprocess OPC UA Data” on page 21-61.

OPC UA Quality

OPC UA Quality values are 32-bit integer values. OPC UA Qualities encode many different characteristics of the quality of the data returned from a current or historical data read operation, including the Major quality (Good, Uncertain, or Bad), quality substatus (dependent on Major quality), value limits (High Limit, Low Limit, Constant), and history origin and characteristics (Raw, Interpolated, Calculated). You can query these characteristics individually using functions specific to the Quality variable that is returned in the read operation. For more information, type

```
help opc.ua.QualityID
```

Working with Time in OPC UA

OPC UA servers return timestamps for server status and for all current and historical read operations. The timestamp represents the time at which the server recorded the data value returned in the read operation. Timestamps are represented in MATLAB by `datetime` values. The datetime values are always returned in the time zone of the MATLAB client used to retrieve the data from the OPC UA server. OPC UA historical read functions require time ranges or specific timestamp arrays over which to retrieve historical data. You can specify time ranges using MATLAB `datetime` values, or as MATLAB date numbers. Any numeric value passed as a timestamp is interpreted as a MATLAB date number. For functions requiring a start and end timestamp, you can also pass a start timestamp and a `duration`.

OPC UA Server Data Types

OPC UA servers store data retrieved from sensors, actuators and other data sources, in Variable Nodes. The Value of each Variable Node is stored and retrieved as a specific Server Data Type, and may be a single value, or an array of values of that data type. The `ServerDataType` property of an `opc.ua.Node` object describes the OPC UA data type used by the server to store the node Value.

When you read data from the server, the value is translated into a corresponding MATLAB data type.

The OPC UA Standard defines simple data types, and Structures which consist of fields containing other data types. Vendors and standards organizations may define extended Data Types, but these are all collections of standard data types, and these collections can be retrieved as multiple Nodes containing Standard Data Types.

The following table describes the OPC UA Standard Data Types, and how these are represented in MATLAB. Any `ServerDataType` value not shown here cannot be read by Industrial Communication Toolbox.

OPC UA Data Type	MATLAB Data Type	Notes
Boolean	Logical	
Byte	uint8	
ByteString (*)	uint8 vector	Arrays converted to cell array of uint8
DateTime (*)	Datetime	
Double	Double	
ExpandedNodeId (*)	Structure	Fields: `NodeId`, `NaspaceUri`, `ServerIndex`
Float	Single	
Guid (*)	Encoded character vector	Arrays converted to cell array of character vectors
Int16	int16	
Int32	int32	
Int64	int64	
LocalizedText	Character vector	Arrays converted to cell array of character vectors
NodeId (*)	Encoded character vector	Arrays converted to cell array of character vectors
QualifiedName (*)	Encoded character vector	Arrays converted to cell array of character vectors
SByte	int8	
String	Character vector	Arrays converted to cell array of character vectors
Structure (*)	Structure	
Time (*)	Datetime	Arrays not supported.
UInt16	uint16	

OPC UA Data Type	MATLAB Data Type	Notes
UInt32	uint32	
UInt64	uint64	
XmlElement (*)	Character vector	Arrays converted to cell array of character vectors

When writing values to an OPC UA server, the value is translated to the equivalent OPC UA Data Type as long as the value is specified as the MATLAB data type described above. You cannot write OPC UA Data Types marked (*).

OPC UA Security

OPC Unified Architecture has been designed to support secure, authenticated connections between OPC UA servers and clients. Nonproprietary, industry standard protocols are used to achieve security in OPC UA communication. Security in OPC UA is provided using three mechanisms:

- Messages passed between an OPC UA client and server can be sent in one of three Message Security Modes:
  - `None`: No security. Messages are sent in clear text.
  - `Sign`: Messages are signed by the sender, to authenticate the origin of the message. However, messages are not encrypted.
  - `SignAndEncrypt`: Messages are signed by the sender, to authenticate the origin of the message, and encrypted to ensure privacy.
- Encryption and signing of the messages is performed using industry standard Asymmetric Cryptography schemes. A Channel Security Policy defines the specific scheme to use for encryption and signing. For a list of currently supported Channel Security Policies in Industrial Communication Toolbox, type the following command in MATLAB:

  ```
  enumeration opc.ua.ChannelSecurityPolicies
  ```

  When setting up a secure connection between the OPC UA Client and OPC UA Server, each of the parties exchange Application Instance Certificates that are used to encrypt and sign messages sent between the parties. These certificates can optionally be checked against a certificate trust list maintained by system administrators for each application to ensure that connections are made to the correct server, from the correct client. Industrial Communication Toolbox currently accepts server certificates automatically when the connection is established. For more information, see “OPC UA Certificate Management” on page 17-9.
- User Authentication may be used by the server to restrict access to features of the server based on the specific user making the connection. Industrial Communication Toolbox supports the following user authentication options:
  - `Anonymous`: A user name is not provided. Some servers might not allow for anonymous user authentication.
  - `Username`: A user name and password combination authenticates the specific user making the connection.
  - `Certificate`: A User Certificate (in X509 standard) is used to authenticate the user. The public key of the certificate must be pre-shared with the server, and when establishing the connection the user must provide the public key, private key, and a password used to protect the private key. Clear (passwordless) private keys are not supported by the toolbox.

Servers normally support more than one security model for clients to use when connecting to the server. The supported security models that a server supports are described through endpoints available from the server. Each endpoint defines one Channel Security Policy, the allowable Message Security Modes, and supported User Authentication types. To use that specific endpoint, the client makes a connection to the endpoint URL provided in the endpoints list and defines the Message Security Mode to use.

You query the available endpoints of a server using `opcuaserverinfo`, or by constructing an OPC UA client with `opcua`. Once you construct an OPC UA client, you can set the security model to use for that connection using `setSecurityModel`. You pass the user credentials when you connect to the server using the `connect` function.

See Also

More About

- “OPC UA Components” on page 17-3
- “OPC UA Certificate Management” on page 17-9

OPC UA Certificate Management

For securing communications between the client and the server, OPC UA relies on certificates exchanged during the connection process. Certificates consist of a private key, held by the owner; a public key, shared with communication partners; and a password to unlock the private key. If a certificate is compromised in any way (for example, by exposing the private key to unknown parties) then the certificate can be placed in a *Revocation List* so that servers know not to trust clients using that certificate.

To ensure that only authorized clients can connect to an OPC UA server, the server administrator might require that any client attempting to connect to the OPC UA server pre-share their Client Application Instance Certificate before a connection can be established. In this case you must export the client public key and the administrator can store that public key in a trust list for the server.

Industrial Communication Toolbox automatically generates a Client Application Instance when you first call `opcuaserverinfo` or construct an OPC UA client with `opcua`. You use `exportClientCertificate` to copy the client public key to a file for sharing with server administrators.

Note for Administrators Currently it is not possible to replace the Client Application Instance Certificate for Industrial Communication Toolbox.

See Also

More About

- “OPC UA Components” on page 17-3
- “OPC UA Security” on page 17-7

OPC UA Aggregate Functions

Introduction

OPC UA servers can return historical data as an aggregate of some function performed on the data history at particular periods. When you request processed data using the `readProcessed` function, you specify an Aggregate to use, and an Aggregate Interval of time over which to perform that Aggregate function. The server then performs the Aggregate function on each period of Aggregate Interval defined, returning one value associated with all the data in that interval. For example, the "Maximum" Aggregate Function returns the maximum value in the Aggregate Interval; the Range Aggregate Function returns the difference between the highest and lowest value in the aggregate interval.

OPC UA Aggregates are represented in MATLAB by a character vector defining the Aggregate Function, or by the `opc.ua.AggregateFnId` enumeration class. For example, to specify that a `readProcessed` operation use the Maximum Aggregate Function, you can use either of the following syntaxes:

```
readProcessed(UaClient,NodeList,'Maximum',...)
readProcessed(UaClient,NodeList,opc.ua.AggregateFnId.Maximum,...)
```

Available Aggregate Functions on an OPC UA Server

When an OPC UA Client is connected to an OPC UA server, the client’s `AggregateFunctions` property stores a list of aggregate functions supported by that server. Servers need not implement every Aggregate Function defined by the OPC UA Standard, but must publish the Aggregate Functions that are supported by that server. Use the `AggregateFunctions` property to ensure that the aggregate function you need is supported by the server. Note, however, that the server might not implement that function for all Variable nodes on the server. If you attempt to retrieve processed data from the server, you might get an "Unsupported Aggregate Function" error, even if the aggregate function is reported as being supported by the server.

OPC UA Standard Aggregate Functions

The following functions are defined by the OPC Foundation.

Function	Description
`AnnotationCount`	Retrieve the number of Annotations in the interval.
`Average`	Retrieve the average value of the data over the interval.
`Count`	Retrieve the number of raw values over the interval.
`Delta`	Retrieve the difference between the `Start` and `End` values in the interval.
`DeltaBounds`	Retrieve the difference between the `StartBound` and `EndBound` values in the interval.
`DurationBad`	Retrieve the total duration of time in the interval during which the data is bad.
`DurationGood`	Retrieve the total duration of time in the interval during which the data is good.

Function	Description
`DurationInStateNonZero`	Retrieve the time a Boolean or numeric was in a nonzero state using Simple Bounding Values.
`DurationInStateZero`	Retrieve the time a Boolean or numeric was in a zero state using Simple Bounding Values.
`End`	Retrieve the value at the end of the interval using Interpolated Bounding Values.
`EndBound`	Retrieve the value at the end of the interval using Simple Bounding Values.
`Interpolative`	At the beginning of each interval, retrieve the calculated value from the data points on either side of the requested timestamp.
`Maximum`	Retrieve the maximum raw value in the interval with the timestamp of the start of the interval.
`Maximum2`	Retrieve the maximum value in the interval including the Simple Bounding Values.
`MaximumActualTime`	Retrieve the maximum value in the interval and the timestamp of the maximum value.
`MaximumActualTime2`	Retrieve the maximum value with the actual timestamp including the Simple Bounding Values.
`Minimum`	Retrieve the minimum raw value in the interval with the timestamp of the start of the interval.
`Minimum2`	Retrieve the minimum value in the interval including the Simple Bounding Values.
`MinimumActualTime`	Retrieve the minimum value in the interval and the timestamp of the minimum value.
`MinimumActualTime2`	Retrieve the minimum value with the actual timestamp including the Simple Bounding Values.
`NumberOfTransitions`	Retrieve the number of changes between zero and nonzero that a Boolean or Numeric value experienced in the interval.
`PercentBad`	Retrieve the percent of data (0 to 100) in the interval which has bad StatusCode.
`PercentGood`	Retrieve the percent of data (0 to 100) in the interval which has good StatusCode.
`Range`	Retrieve the difference between the `Minimum` and `Maximum` values over the interval.
`Range2`	Retrieve the difference between the `Minimum2` and `Maximum2` values over the interval.
`StandardDeviationPopulation`	Retrieve the standard deviation for the interval for a complete population (n) which includes Simple Bounding Values.
`StandardDeviationSample`	Retrieve the standard deviation for the interval for a sample of the population (n-1).
`Start`	Retrieve the value at the beginning of the interval using Interpolated Bounding Values.

Function	Description
`StartBound`	Retrieve the value at the beginning of the interval using Simple Bounding Values.
`TimeAverage`	Retrieve the time weighted average data over the interval using Interpolated Bounding Values.
`TimeAverage2`	Retrieve the time weighted average data over the interval using Simple Bounding Values.
`Total`	Retrieve the total (time integral) of the data over the interval using Interpolated Bounding Values.
`Total2`	Retrieve the total (time integral) of the data over the interval using Simple Bounding Values.
`VariancePopulation`	Retrieve the variance for the interval as calculated by the `StandardDeviationPopulation` which includes Simple Bounding Values.
`VarianceSample`	Retrieve the variance for the interval as calculated by the `StandardDeviationSample`.
`WorstQuality`	Retrieve the worst StatusCode of data in the interval.
`WorstQuality2`	Retrieve the worst StatusCode of data in the interval including the Simple Bounding Values.

Access Data from OPC UA Servers

In this section...

"OPC UA Programming Overview" on page 17-13
"Step 1: Locate Your OPC UA Server" on page 17-13
"Step 2: Create an OPC UA Client and Connect to the Server" on page 17-14
"Step 3: Browse OPC UA Server Namespace" on page 17-15
"Step 4: Read Current Values from the OPC UA Server" on page 17-16
"Step 5: Read Historical Data from the OPC UA Server" on page 17-17
"Step 6: Plot the Data" on page 17-18
"Step 7: Clean Up" on page 17-18

OPC UA Programming Overview

This topic shows the basic steps to create an OPC Unified Architecture (UA) application by retrieving current and historical data from a Simulation Server running on your local machine.

Note To run the sample code in the following steps you need the Prosys OPC UA Simulation Server running on your local machine. You can also optionally install the Local Discovery Service and register the Prosys server with the LDS. For installation details, see "Install an OPC UA Simulation Server for OPC UA Examples" on page 1-15. The code requires only minor changes to work with other servers.

Step 1: Locate Your OPC UA Server

In this step, you obtain information that the toolbox needs to uniquely identify the OPC UA server that you want to connect to. You use this information when creating an OPC UA client object, described in Step 2: Create an OPC UA Client Object.

The first piece of information is the host name of the server computer. The host name (a descriptive name like "HistorianServer" or an IP address such as 192.168.16.32) qualifies that computer on the network, and is used by the OPC protocols to determine the available OPC servers on that computer. In any OPC application, you must know the name of the OPC server host, so that a connection with that host can be established. Your network administrator can provide a list of host names that provide OPC servers on your network. In this example, you will use `'localhost'` as the host name, because you will connect to the OPC server on the same machine as the client.

OPC UA servers are uniquely identified by Universal Resource Locations. Similar to web addresses, a URL for an OPC UA server starts with `opc.tcp://`, and then provides an address of the server as a hostname, port, and address in standard ASCII text. For example, the URL for the Prosys OPC UA Simulation Server is `opc.tcp://localhost:53530/OPCUA/SimulationServer`.

OPC UA Server URLs are advertised through an OPC UA Local Discovery Service (LDS), available on every OPC UA server host machine. Your system administrator can provide a list of server URLs for a particular host, or you can query the host for all available OPC UA servers.

If you have installed the LDS and registered the OPC UA server with the LDS, you can use the `opcuaserverinfo` function to query hosts from the command line. If you have not installed the LDS, skip to Step 2.

```
serverList = opcuaserverinfo('localhost')
```

```
serverList =
1x2 OPC UA ServerInfo array:
    index         Description               Hostname    Port
    -----  ----------------------------  ------------  -----
      1    SimulationServer              myhost.local  53530
      2    Quickstart Data Access Server myhost.local  62547
```

Locate the server of interest by using the `findDescription` function to search for a specific character vector in the server description.

```
hsInfo = findDescription(serverList,'Simulation')
```

```
hsInfo =
OPC UA ServerInfo 'SimulationServer':

   Connection Information
    Hostname: 'myhost.local'
        Port: 53530
```

From this discovery process, you can identify the port (53530) on which the OPC UA server listens for connections. The discovery process also makes it easier to construct and connect to the required OPC UA server.

Step 2: Create an OPC UA Client and Connect to the Server

After locating your OPC UA server, you create an OPC UA Client to manage the connection to the server, obtain key server characteristics, and read and write data from the server. You can use the `opcuaserverinfo` result to construct an OPC UA client directly.

```
uaClient = opcua(hsInfo)
```

Or you could create a client using the hostname and port directly.

```
uaClient = opcua('localhost',53530)
```

```
uaClient =
OPC UA Client:

   Server Information:
                     Name: 'SimulationServer@localhost'
                 Hostname: 'localhost'
                     Port: 53530
              EndpointUrl: 'opc.tcp://localhost:53530/OPCUA/SimulationServer'

   Connection Information:
                  Timeout: 10
                   Status: 'Disconnected'
              ServerState: '<Not connected>'

   Security Information:
      MessageSecurityMode: SignAndEncrypt
    ChannelSecurityPolicy: Aes256_Sha256_RsaPss
                Endpoints: [1×11 opc.ua.EndpointDescription]
```

The client is initially disconnected from the server, as shown by the `Status` property. After you connect to the server, additional properties are shown in the client display.

```
connect(uaClient)
```

```
uaClient

OPC UA Client:

   Server Information:
                      Name: 'SimulationServer@localhost'
                  Hostname: 'localhost'
                      Port: 53530
               EndpointUrl: 'opc.tcp://localhost:53530/OPCUA/SimulationServer'

   Connection Information:
                   Timeout: 10
                    Status: 'Connected'
               ServerState: 'Running'

   Security Information:
        MessageSecurityMode: SignAndEncrypt
      ChannelSecurityPolicy: Aes256_Sha256_RsaPss
                 Endpoints: [1×11 opc.ua.EndpointDescription]

   Server Limits:
             MinSampleRate: 0 sec
              MaxReadNodes: 0
             MaxWriteNodes: 0
        MaxHistoryReadNodes: 0
    MaxHistoryValuesPerNode: 0
```

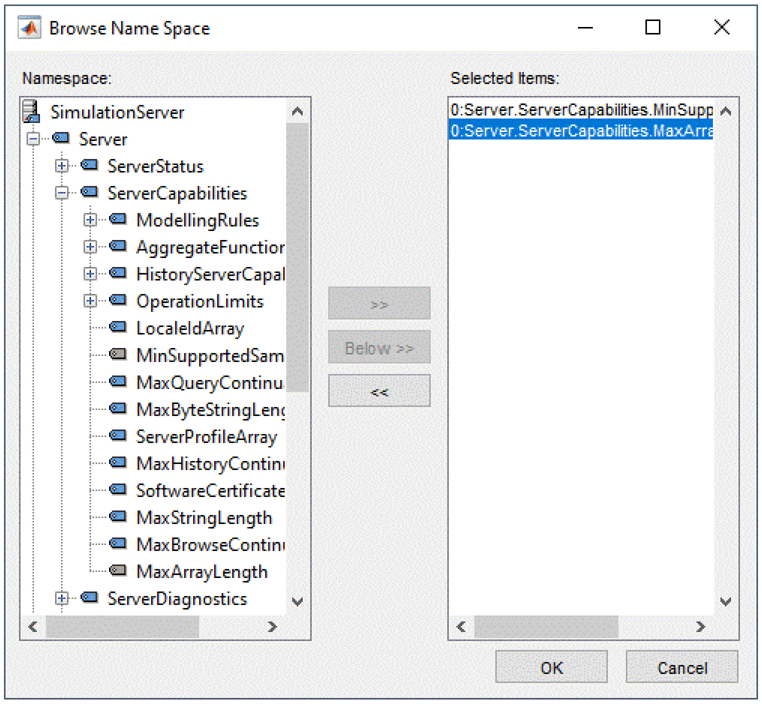
The additional properties describe capabilities of the server, notably limits for various read and write operations. A limit value of 0 indicates that the server does not impose a direct limit on that capability.

Step 3: Browse OPC UA Server Namespace

OPC UA servers provide a single namespace for you to read and write both current data and historical data. The namespace is organized as a hierarchy of nodes. Each node has attributes which describe that node. A node is uniquely identified by two elements: A namespace index (numeric integer) and a node identifier (numeric integer, character vector, or Globally Unique Identifier or GUID). To uniquely describe a node, you have to provide both the namespaceindex and the identifier; you cannot provide only the identifier because that might be repeated for different namespace indexes.

Industrial Communication Toolbox exposes the hierarchy of nodes through the namespace property of the OPC UA client. Each element of the namespace property is a node at the top-most level of the server. Every node in the namespace has a `Children` property which exposes the subnodes contained in that node. You can browse the namespace graphically using the `browseNamespace` function. The resulting dialog box allows you to select nodes from the hierarchy and return them in the output from the function.

```
serverNodes = browseNamespace(uaClient)
```



When you click **OK** the selected items are returned in the command window output.

```
serverNodes = 
1x2 OPC UA Node array:
    index           Name           NsInd  Identifier  NodeType  Children
    -----  ---------------------  -----  ----------  --------  --------
      1    MinSupportedSampleRate  0      2272        Variable  0
      2    MaxArrayLength          0      11702       Variable  0
```

Nodes can have data values associated with them or can simply be containers for other nodes. The `NodeType` property of a node identifies the node as an object node (container) or variable node (data). For more information on how to programmatically search the server namespace, see “Browse OPC UA Server Namespace” on page 21-44.

Step 4: Read Current Values from the OPC UA Server

OPC UA servers provide access to both current and historical values of their `Variable` nodes. With Industrial Communication Toolbox you use arrays of Nodes to read current values from the server. Current data includes the value, a timestamp that the server received the data value from the sensor, and a quality describing the accuracy and source of the data value.

```
[val,ts,qual] = readValue(uaClient,serverNodes)
```

```
val = 
  2×1 cell array
    {[0 sec]}
    {[65535]}
ts = 
  2×1 datetime array
   10-Apr-2019 09:46:43
   10-Apr-2019 09:46:43
```

```
qual =
OPC UA Quality ID:
    'Good'
    'Good'
```

For more information on reading and writing current values, see “Read and Write Current OPC UA Server Data” on page 21-51.

Step 5: Read Historical Data from the OPC UA Server

Historical data is stored for selected nodes on the OPC UA server. The server nodes retrieved in the previous step will not be archived by the server because the values do not generally change. You can query the `Historizing` property of a Node to determine if the server is currently archiving data for that node.

Because the serverNode list is an array, you must collect the outputs using concatenation.

```
[serverNodes.Historizing]
```

```
ans =
     0     0
```

None of the server nodes are currently being historized. In addition, the server does not allow historical access to these nodes, as evidenced by the `AccessLevelHistory` property of the nodes.

```
{serverNodes.AccessLevelHistory}
```

```
ans =
    'none'    'none'
```

To locate nodes with history, query the server for the Double and Int32 nodes in the `Simulation` parent node.

```
simNode = findNodeByName(uaClient.Namespace,'Simulation')
```

```
simNode = 

OPC UA Node:

   Node Information:
                      Name: 'Simulation'
               Description: 'The type for objects that organize other nodes.'
            NamespaceIndex: 5
                Identifier: '85/0:Simulation'
                  NodeType: 'Object'

   Hierarchy Information:
                    Parent: Server
                  Children: 14
```

The `Simulation` node is an `Object` node, so it has no `Value`. However, it has 7 Children. Locate the `Sinusoid` and `Random` child nodes. The `'-partial'` flag finds nodes beginning with the argument provided.

```
sineNode = findNodeByName(simNode,'Sinusoid', '-partial');
randNode = findNodeByName(simNode,'Random', '-partial')
```

```
randNode = 

OPC UA Node:

   Node Information:
                      Name: 'Random1'
```

```
              Description: ''
           NamespaceIndex: 5
               Identifier: 'Random1'
                 NodeType: 'Variable'

   Hierarchy Information:
                   Parent: 'Simulation'
                 Children: 0


           ServerDataType: Double
       AccessLevelCurrent: read/write
       AccessLevelHistory: read
              Historizing: 0
```

Although the `Sinusoid1` and `Random1` nodes are not currently being archived (`Historizing` is false) you can read history data from the nodes (the history was logged at startup, and then turned off). To read all data stored on the server within a specified time range, use the `readHistory` function, passing the nodes to read and the time range over which to read the data.

```
histData = readHistory(uaClient,[sineNode,randNode],datetime('now')-seconds(10),datetime('now'))
```

```
histData =
1-by-2 OPC UA Data object array:
           Timestamp                 Sinusoid1                 Random1
    -----------------------  -------------------------  -------------------------
    2019-04-10 09:58:31.000        0.415823 [Good (Raw)]        0.131428 [Good (Raw)]
    2019-04-10 09:58:32.000        0.813473 [Good (Raw)]        0.038980 [Good (Raw)]
    2019-04-10 09:58:33.000        1.175570 [Good (Raw)]        0.316324 [Good (Raw)]
    2019-04-10 09:58:34.000        1.486290 [Good (Raw)]        0.229609 [Good (Raw)]
    2019-04-10 09:58:35.000        1.732051 [Good (Raw)]        0.208826 [Good (Raw)]
    2019-04-10 09:58:36.000        1.902113 [Good (Raw)]        0.483303 [Good (Raw)]
    2019-04-10 09:58:37.000        1.989044 [Good (Raw)]        0.393722 [Good (Raw)]
    2019-04-10 09:58:38.000        1.989044 [Good (Raw)]        0.206232 [Good (Raw)]
    2019-04-10 09:58:39.000        1.902113 [Good (Raw)]        0.116650 [Good (Raw)]
    2019-04-10 09:58:40.000        1.732051 [Good (Raw)]        0.391128 [Good (Raw)]
```

Obtain a summary of the data retrieved.

```
summary(histData)
```

```
1-by-2 OPC UA Data object:
      Name          Value            Start Timestamp           End Timestamp                Quality
    ---------  ----------------  -----------------------  -----------------------  -----------------------------
    Sinusoid1  10 double values  2019-04-10 09:58:31.000  2019-04-10 09:58:40.000  1 unique quality [Good (Raw)]
    Random1    10 double values  2019-04-10 09:58:31.000  2019-04-10 09:58:40.000  1 unique quality [Good (Raw)]
```

Step 6: Plot the Data

You can plot the data directly from the resulting `opc.ua.Data` object.

```
plot(histData)
legend show
```

You can also convert the data into MATLAB native data types for further processing. For information on processing data, see “Visualize and Preprocess OPC UA Data” on page 21-61.

Step 7: Clean Up

When you have finished exchanging data with the OPC server, you should disconnect from the server.

```
disconnect(uaClient)
```

You can then clear the OPC UA variables from MATLAB memory. If you clear an OPC UA client from memory, the connection to the server is automatically closed.

Non-OPC Technologies

Controlling Devices Using Modbus

- “Modbus Interface Supported Features” on page 18-2
- “Create a Modbus Connection” on page 18-3
- “Configure Properties for Modbus Communication” on page 18-5
- “Read Data from a Modbus Server” on page 18-8
- “Read Temperature from a Remote Temperature Sensor” on page 18-13
- “Write Data to a Modbus Server” on page 18-14
- “Write and Read Multiple Holding Registers” on page 18-16
- “Modify the Contents of a Holding Register Using a Mask Write” on page 18-18
- “Use the Modbus Explorer App” on page 18-19
- “Configure a Connection in the Modbus Explorer” on page 18-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 18-26
- “Control a PLC Using the Modbus Explorer” on page 18-28
- “Generate a Script from Your Modbus Explorer Session” on page 18-33
- “Troubleshooting the Modbus Interface” on page 18-36

Modbus Interface Supported Features

In this section...
"Modbus Capabilities" on page 18-2
"Supported Platforms for Modbus" on page 18-2

Modbus Capabilities

Industrial Communication Toolbox supports the Modbus interface over TCP/IP or Serial RTU. You can use it to communicate with Modbus servers, such as controlling a PLC (Programmable Logic Controller), communicating with a temperature controller, controlling a stepper motor, sending data to a DSP, reading bulk memory from a PAC controller, or monitoring temperature and humidly on a Modbus probe.

Using the Modbus interface, you can do the following tasks, which correspond to the Modbus function codes listed in the table.

Functionality	Modbus Function Code
Read and write coils	1, 5, 15
Read discrete inputs	2
Read and write holding registers	3, 6, 16
Read input registers	4
Perform mask writes on holding registers	22
Perform write/read (in one operation) on holding registers	23

Supported Platforms for Modbus

Industrial Communication Toolbox supports the Modbus interface over TCP/IP or Serial RTU. It is supported on the following platforms.

- Linux® 64-bit
- Mac OS 64-bit
- Microsoft Windows 64-bit

Create a Modbus Connection

Industrial Communication Toolbox supports the Modbus interface over TCP/IP or Serial RTU. You can use it to communicate with Modbus servers, such as a PLC. The typical workflow is:

- Create a Modbus connection to a server or hardware.
- Configure the connection if necessary.
- Perform read and write operations, such as communicating with a temperature controller.
- Clear and close the connection.

To communicate over the Modbus interface, you first create a Modbus object using the `modbus` function. Creating the object also makes the connection. The syntax is:

```
<objname> = modbus('Transport','DeviceAddress')
```

or

```
<objname> = modbus('Transport','Port')
```

You must set the transport type as either `'tcpip'` or `'serialrtu'` to designate the protocol you want to use. Then set the address and port, as shown in the next sections. You can also use arguments in the object creation to set properties such as `Timeout` and `ByteOrder`.

When you create the Modbus object, it connects to the server or hardware. If the transport is `'tcpip'`, then `DeviceAddress` must be specified. Port is optional and defaults to 502 (reserved port for Modbus). If the transport is `'serialrtu'`, then `'Port'` must be specified.

Create Object Using TCP/IP Transport

When the transport is `'tcpip'`, you must specify `DeviceAddress`. This is the IP address or host name of the Modbus server. `Port` is the remote port used by the Modbus server. Port is optional and defaults to `502`, which is the reserved port for Modbus.

This example creates the Modbus object `m` using the device address shown and `port` of `308`.

```
m = modbus('tcpip', '192.168.2.1', 308)
```

```
m = 

   Modbus TCPIP with properties:

    DeviceAddress: '192.168.2.1'
             Port: 308
           Status: 'open'
       NumRetries: 1
          Timeout: 10 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

Create Object Using Serial RTU Transport

When the transport is `'serialrtu'`, you must specify `'Port'`. This is the serial port the Modbus server is connected to.

This example creates the Modbus object `m` using `port 'COM3'`.

```
m = modbus('serialrtu','COM3')
```

```
m = 

   Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 10 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

Create an Object with a Property Setting

You can create the object using a name-value pair to set properties such as `Timeout`. The `Timeout` property specifies the maximum time in seconds to wait for a response from the Modbus server, and the default is `10`. You can change the value either during object creation or after you create the object.

For the list and description of properties you can set for both transport types, see "Configure Properties for Modbus Communication" on page 18-5.

This example creates a Modbus object using Serial RTU, with an increased `Timeout` of `20` seconds.

```
m = modbus('serialrtu','COM3','Timeout'=20)
```

```
m = 

   Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 20 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

The object display in the output shows the specified `Timeout` property value.

Configure Properties for Modbus Communication

The `modbus` object has the following properties.

Property	Transport Type	Description
`'DeviceAddress'`	TCP/IP only	IP address or host name of Modbus server, for example, `'192.168.2.1'`. Required during object creation if transport is TCP/IP. `m = modbus('tcpip', '192.168.2.1')`
`Port`	TCP/IP only	Remote port used by Modbus server. Optional: default is 502. `m = modbus('tcpip', '192.168.2.1', 308)`
`'Port'`	Serial RTU only	Serial port that Modbus server is connected to, for example, `'COM1'`. Required during object creation if transport is Serial RTU. `m = modbus('serialrtu','COM3')`
`Timeout`	TCP/IP and Serial RTU	Maximum time in seconds to wait for a response from the Modbus server, specified as a positive value of type `double`. The default is `10`. You can set the value during object creation or afterward. `m.Timeout = 30;`
`'NumRetries'`	TCP/IP and Serial RTU	Number of retries to perform if there is no reply from the server after a timeout. If using the Serial RTU transport, the message is resent. If using the TCP/IP transport, the connection is closed and reopened. `m.NumRetries = 5;`
`'ByteOrder'`	TCP/IP and Serial RTU	Byte order of values written to or read from 16-bit registers. Valid choices are `'big-endian'` and `'little-endian'`. The default is `'big-endian'`, as specified by the Modbus standard. `m.ByteOrder = 'little-endian';`
`'WordOrder'`	TCP/IP and Serial RTU	Word order for register reads and writes that span multiple 16-bit registers. Valid choices are `'big-endian'` and `'little-endian'`. The default is `'big-endian'`, and it is device-dependent. `m.WordOrder = 'little-endian';`
`'BaudRate'`	Serial RTU only	Bit transmission rate for serial port communication. Default is 9600 bits per second, but the actual required value is device-dependent. `m.Baudrate = 28800;`

Property	Transport Type	Description
'DataBits'	Serial RTU only	Number of data bits to transmit. Default is 8, which is the Modbus standard for Serial RTU. Other valid values are 5, 6, and 7. `m.DataBits = 6;`
'Parity'	Serial RTU only	Type of parity checking. Valid options are `'none'` (default), `'even'`, `'odd'`, `'mark'`, and `'space'`. The actual required value is device-dependent. If set to the default of `none`, parity checking is not performed, and the parity bit is not transmitted. `m.Parity = 'odd';`
'StopBits'	Serial RTU only	Number of bits to indicate the end of data transmission. Valid choices are `1` (default) and `2`. Actual required value is device-dependent, though `1` is typical for even/odd parity, and `2` for no parity. `m.StopBits = 2;`

Set a Property During Object Creation

You can change property values either during object creation or after you create the object. To set property values during object creation, specify name-value pair arguments to the `modbus` function call.

This example creates a Modbus object and increases the `Timeout` to `20` seconds.

```
m = modbus('serialrtu','COM3','Timeout'=20)
```

```
m = 

   Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 20 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

The object display in the output shows the specified `Timeout` property value.

Set a Property After Object Creation

You can change a property value on an existing object with this syntax.

```
<object_name>.<property_name> = <property_value>
```

This example using the Modbus object `m`, increases the `Timeout` to 30 seconds.

```
m = modbus('serialrtu','COM3');
m.Timeout = 30
```

This example changes the `Parity` from the default of `'none'` to `'even'`.

```
m = modbus('serialrtu','COM3');
m.Parity = 'even';
```

Read Data from a Modbus Server

In this section...

“Types of Data You Can Read over Modbus” on page 18-8
“Read Coils over Modbus” on page 18-8
“Read Inputs over Modbus” on page 18-9
“Read Input Registers over Modbus” on page 18-9
“Read Holding Registers over Modbus” on page 18-10
“Specify Server ID and Precision” on page 18-10
“Read Mixed Data Types” on page 18-11

Types of Data You Can Read over Modbus

The `read` function performs read operations from four types of target-addressable areas:

- Coils
- Inputs
- Input registers
- Holding registers

When you perform the read, you must specify the target type (`target`), the starting address (`address`), and the number of values to read (`count`). You can also optionally specify the address of the server (`serverId`) for any target type, and the data format (`precision`) for registers.

For an example showing the entire workflow of reading a holding register on a PLC, see “Read Temperature from a Remote Temperature Sensor” on page 18-13.

Read Coils over Modbus

If the read target is coils, the function reads the values from 1–2000 contiguous coils in the remote server, starting at the specified address. A coil is a single output bit. A value of `1` indicates the coil is on and a value of `0` means it is off.

The syntax to read coils is:

```
read(obj,'coils',address,count)
```

The `obj` parameter is the name of the Modbus object. The following examples assume you have created a Modbus object, `m`. For information on creating the object, see “Create a Modbus Connection” on page 18-3.

The `address` parameter is the starting address of the coils to read, specified as a double. The `count` parameter is the number of coils to read, specified as a double. If the read is successful, it returns a vector of double values of `1` or `0`, where the first value in the vector corresponds to the coil value at the starting address.

This example reads 8 coils, starting at address 1.

```
read(m,'coils',1,8)
```

```
ans =

   1   1   0   1   1   0   1   0
```

You can also read values into a variable for later access.

```
data = read(m,'coils',1,8)
```

```
data =

   1   1   0   1   1   0   1   0
```

Read Inputs over Modbus

If the read target is inputs, the function reads the values from 1–2000 contiguous discrete inputs in the remote server, starting at the specified address. A discrete input is a single input bit. A value of `1` indicates the input is on, and a value of `0` means it is off.

The syntax to read inputs is:

```
read(obj,'inputs',address,count)
```

The `obj` parameter is the name of the Modbus object. The following examples assume you have created a Modbus object, `m`. For information on creating the object, see "Create a Modbus Connection" on page 18-3.

The `address` parameter is the starting address of the inputs to read, specified as a double. The `count` parameter is the number of inputs to read, specified as a double. If the read operation is successful, it returns a vector of double values of `1` or `0`, where the first value in the vector corresponds to the input value at the starting address.

This example reads 10 discrete inputs, starting at address 2.

```
read(m,'inputs',2,10)
```

```
ans =

   1   1   0   1   1   0   1   0   0   1
```

Read Input Registers over Modbus

If the read target is input registers, the function reads the values from 1–125 contiguous input registers in the remote server, starting at the specified address. An input register is a 16-bit read-only register.

The syntax to read input registers is:

```
read(obj,'inputregs',address,count)
```

The `obj` parameter is the name of the Modbus object. The following examples assume you have created a Modbus object, `m`. For information on creating the object, see "Create a Modbus Connection" on page 18-3.

The `address` parameter is the starting address of the input registers to read, specified as a double. The `count` parameter is the number of input registers to read, specified as a double. If the read operation is successful, it returns a vector of doubles. Each double represents a 16-bit register value, where the first value in the vector corresponds to the input register value at the starting address.

This example reads 4 input registers, starting at address 20.

```
read(m,'inputregs',20,4)
```

```
ans =

   27640   60013   51918   62881
```

Read Holding Registers over Modbus

If the read target is holding registers, the function reads the values from 1–125 contiguous holding registers in the remote server, starting at the specified address. A holding register is a 16-bit read/write register.

The syntax to read inputs is:

```
read(obj,'holdingregs',address,count)
```

The `obj` parameter is the name of the Modbus object. The following examples assume you have created a Modbus object, `m`. For information on creating the object, see “Create a Modbus Connection” on page 18-3.

The `address` parameter is the starting address of the holding registers to read,specified as a double. The `count` parameter is the number of holding registers to read, specified as a double. If the read operation is successful, it returns a vector of doubles. Each double represents a 16-bit register value, where the first value in the vector corresponds to the holding register value at the starting address.

This example reads 4 holding registers, starting at address 20.

```
read(m,'holdingregs',20,4)
```

```
ans =

   27640   60013   51918   62881
```

For an example showing the entire workflow of reading a holding register on a PLC, see “Read Temperature from a Remote Temperature Sensor” on page 18-13.

Specify Server ID and Precision

You can read any of the four types of targets and also specify the optional parameters for server ID, and can specify precision for registers.

Server ID Option

The `serverId` argument specifies the address of the server to send the read command to. Valid values are `0`–`247`, with `0` being the broadcast address. This argument is optional, and the default is `1`.

Note If your device uses a `slaveID` property, it might work to use it as the `serverID` property with the `read` command as described here.

The syntax to specify server ID is:

```
read(obj,target,address,count,serverId)
```

This example reads 8 coils starting at address 1 from server ID 3.

```
read(m,'coils',1,8,3)
```

Precision Option

The `'precision'` argument specifies the data format of the register being read on the Modbus server. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

Note that `'precision'` does not refer to the return type, which is always `'double'`. It only specifies how to interpret the register data.

The syntax to specify precision is:

```
read(obj,target,address,count,precision)
```

This example reads 8 holding registers starting at address 1 using a precision of `'uint32'`.

```
read(m,'holdingregs',1,8,'uint32')
```

Both Options

You can set both the `serverId` option and the `'precision'` option together when the target is a register. When you use both options, the `serverId` is listed first after the required arguments.

The syntax to specify both Server ID and precision is:

```
read(obj,target,address,count,serverId,precision)
```

This example reads 8 holding registers starting at address 1 using a precision of `'uint32'` from Server ID 3.

```
read(m,'holdingregs',1,8,3,'uint32')
```

Read Mixed Data Types

You can read contiguous values of different data types (precisions) in a single read operation by specifying the data type for each value. You can do that within the syntax of the `read` function, or set up variables containing arrays of counts and precisions. Both methods are shown here.

Within the `read` Syntax

An example of the `read` function with one data type is:

```
read(m,'holdingregs',500,10,'uint32')
```

In that example, the target type is holding registers, the starting address is `500`, the count is `10`, and the precision is `uint32`. To read 10 values of mixed data types, you can use this syntax:

```
read(m,'holdingregs',500,[3 2 3 2],{'uint16','single','double','int16'})
```

You use arrays to specify both counts and precisions. In this case, the counts are 3, 2, 3, and 2. The function reads 3 values of data type `uint16`, 2 values of data type `single`, 3 values of data type double, and 2 values of data type int16. The registers are contiguous, starting at address 500. This example reads 3 uint16 values from addresses 500-502, 2 single values from addresses 503-506, 3 double values from addresses 507-518, and 2 int16 values from addresses 519-520, all in one operation.

Use Variables

Instead of using literal arrays inside the read function as shown above, you can specify variables with array values as function arguments. The equivalent for the example shown above is:

```
count = [3 2 3 2]
precision = {'uint16','single','double','int16'}
read(m,'holdingregs',500,count,precision)
```

Using variables is convenient when you have a lot of values to read and they are of mixed data types.

Read Temperature from a Remote Temperature Sensor

This example shows how to read temperature and humidity measurements from a remote sensor on a PLC connected via TCP/IP. The temperature sensor is connected to a holding register at address 1 on the board, and the humidity sensor is at address 5.

1 Create the Modbus object, using TCP/IP.

```
m = modbus('tcpip', '192.168.2.1', 502)
```

```
m = 

   Modbus TCPIP with properties:

    DeviceAddress: '192.168.2.1'
             Port: 502
           Status: 'open'
       NumRetries: 1
          Timeout: 10 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

2 The humidity sensor does not always respond instantly, so increase the timeout value to 20 seconds.

```
m.Timeout = 20
```

3 The temperature sensor is connected to a holding register at address 1 on the board. Read one value to get the current temperature reading. As the temperature value is a double, set the precision to double.

```
read(m,'holdingregs',1,1,'double')
```

```
ans =

   46.7
```

4 The humidity sensor is connected to the holding register at address 5 on the board. Read one value to get the current humidity reading.

```
read(m,'holdingregs',5,1,'double')
```

```
ans =

   35.8
```

5 Disconnect from the server and clear the object.

```
clear m
```

Write Data to a Modbus Server

In this section...

"Types of Data You Can Write to over Modbus" on page 18-14
"Write Coils over Modbus" on page 18-14
"Write Holding Registers over Modbus" on page 18-14

Types of Data You Can Write to over Modbus

The `write` function performs write operations to two types of target addressable areas:

- Coils
- Holding registers

Each of the two areas can accept a write request to a single address or a contiguous address range. When you perform the write operation, you must specify the target type (`target`), the starting address (`address`), and the values to write (`values`). You can also optionally specify the address of the server (`serverId`) and the data format (`precision`).

Write Coils over Modbus

If the write target is coils, the function writes a contiguous sequence of 1–1968 coils to either on or off (1 or 0) in a remote device. A coil is a single output bit. A value of `1` indicates the coil is on, and a value of `0` means it is off.

The syntax to write to coils is:

```
write(obj,'coils',address,values)
```

The `obj` parameter is the name of the Modbus object. The following examples assume you have created a Modbus object, `m`. For information on creating the object, see "Create a Modbus Connection" on page 18-3.

The `address` parameter is the starting address of the coils to write to, specified as a double. The `values` parameter is an array of values to write. For a target of coils, valid values are `0` and `1`.

This example writes to 4 coils, starting at address 8289.

```
write(m,'coils',8289,[1 1 0 1])
```

You can also create a variable for the values to write.

```
values = [1 1 0 1];
write(m,'coils',8289,values)
```

Write Holding Registers over Modbus

If the write target is holding registers, the function writes a block of 1–123 contiguous registers in a remote device. Values whose representation is greater than 16 bits are stored in consecutive register addresses.

The syntax to write to holding registers is:

```
write(obj,'holdingregs',address,values)
```

The `obj` parameter is the name of the Modbus object. The following examples assume you have created a Modbus object, `m`. For information on creating the object, see “Create a Modbus Connection” on page 18-3.

The `address` parameter is the starting address of the holding registers to write to, specified as a double. The `values` parameter is an array of values to write. For a target of holding registers, valid values must be in the range of the specified precision.

This example sets the register at address 49153 to 2000.

```
write(m,'holdingregs',49153,2000)
```

Precision Option

The `'precision'` argument specifies the data format of the register being written to on the Modbus server. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

The values passed in to be written are converted to register values based on the specified precision. For precision values `'int32'`, `'uint32'`, and `'single'`, each value corresponds to two registers, and for `'uint64'`, `'int64'` and `'double'`, each value corresponds to four registers. For `'int16'` and `'uint16'`, each value is from one 16-bit register.

This example writes 3 values as `single` precision, starting at address 29473.

```
write(m,'holdingregs',29473,[928.1 50.3 24.4],'single')
```

Write and Read Multiple Holding Registers

The `writeRead` function performs a combination of one write operation and one read operation on groups of holding registers in a single Modbus transaction. The write operation is always performed before the read. The range of addresses to read and the range of addresses to write must be contiguous, but each is specified independently and they can overlap.

The syntax for the write-read operation to holding registers is:

```
writeRead(obj,writeAddress,values,readAddress,readCount)
```

The `obj` parameter is the name of the Modbus object. The following examples assume you have created a Modbus object, `m`. For information on creating the object, see “Create a Modbus Connection” on page 18-3.

The `writeAddress` is the starting address of the holding registers to write to, specified as a double. The `values` parameter is an array of values to write. The first value in the vector is written to the `writeAddress`. Each value must be in the range `0–65535`.

The `readAddress` is the starting address of the holding registers to read, and `readCount` is the number of registers to read.

If the operation is successful, it returns an array of doubles, each representing a 16-bit register value, where the first value in the vector corresponds to the register value at the address specified in `readAddress`.

This example writes two holding registers starting at address 601, and reads 4 holding registers starting at address 19250.

```
writeRead(m,601,[1024 512],19250,4)
```

```
ans =

   27640   60013   51918   62881
```

You can optionally create variables for the values to be written, instead of including the literal array of values in the function syntax. The same example could be written this way, using a variable for the write values:

```
values = [1024 512];
writeRead(m,601,values,19250,4)
```

```
ans =

   27640   60013   51918   62881
```

Server ID Option

The `serverId` argument specifies the address of the server to send the read command to. Valid values are `0–247`, with `0` being the broadcast address. This argument is optional, and the default is `1`.

Note If your device uses a `slaveID` property, it might work to use it as the `serverID` property with the `writeRead` command as described here.

The syntax to specify a server ID is:

```
writeRead(obj,writeAddress,values,readAddress,readCount,serverId)
```

This example writes 3 holding registers starting at address 400, and reads 4 holding registers starting at address 52008, from server ID 6.

```
writeRead(m,400,[1024 512 680],52008,4,6)
```

```
ans =

   38629   84735   29456   39470
```

Precision Option

The `'writePrecision'` and `'readPrecision'` arguments specify the data format of the register being read from or written to on the Modbus server. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

The values passed in to be written are converted to register values based on the specified precision. For precision values `'int32'`, `'uint32'`, and `'single'`, each value corresponds to two registers, and for `'uint64'`, `'int64'` and `'double'`, each value corresponds to four registers. For `'int16'` and `'uint16'`, each value is from one 16-bit register.

Note that precision specifies how to interpret or convert the register data, not the return type of the read operation. The data returned is always of type double.

The syntax for designating the write and read precision is:

```
writeRead(obj,writeAddress,values,writePrecision,readAddress,readCount,readPrecision)
```

If you want to use the `serverId` argument as well, it goes after the `readPrecision`.

This example writes 3 holding registers starting at address 400 and reads 4 holding registers starting at address 52008, from server ID 6. It also specifies a `writePrecision` of `'uint64'` and a `readPrecision` of `'uint32'`.

```
writeRead(m,400,[1024 512 680],'uint64',52008,4,'uint32',6)
```

```
ans =

   38629   84735   29456   39470
```

This example reads two holding registers starting at address 919, and writes 3 holding registers starting at address 719, formatting the read and write values for single precision data registers.

```
  values = [1.14 5.9 11.27];
  writeRead(m,719,values,'single',919,2,'single')
```

Modify the Contents of a Holding Register Using a Mask Write

You can modify the contents of a holding register using the `maskWrite` function. The function can set or clear individual bits in a specific holding register. It performs a read/modify/write operation, using a combination of an AND mask, an OR mask, and the current contents of the register.

The function algorithm works as follows:

```
 Result = (register value AND andMask) OR (orMask AND (NOT andMask))
```

For example:

```
                 Hex    Binary
Current contents  12   0001 0010
And_Mask          F2   1111 0010
Or_Mask           25   0010 0101
(NOT And_Mask)    0D   0000 1101
                  --   ----------
Result            17   0001 0111
```

If the `orMask` value is 0, the result is simply the logical ANDing of the current contents and the `andMask`. If the `andMask` value is 0, the result is equal to the `orMask` value.

The contents of the register can be read by using the `read` function with the target set to `'holdingregs'`. However, the contents values could be changed subsequently as the controller scans its user logic program.

The syntax for the mask write operation for holding registers is:

```
maskWrite(obj, address, andMask, orMask)
```

If you want to designate a server ID, use:

```
maskWrite(obj, address, andMask, orMask, serverId)
```

The `obj` parameter is the name of the Modbus object. The following examples assume you have created a Modbus object, `m`. For information on creating the object, see “Create a Modbus Connection” on page 18-3.

The `address` is the register address to perform mask write on. The `andMask` argument is the `AND` value to use in the mask write operation. The valid range is `0–65535`. The `orMask` argument is the `OR` value to use in the mask write operation. The valid range is `0–65535`.

This example establishes bit 0 at address 20, and performs a mask write operation. Because the `andMask` is 6, that clears all bits except for bits 1 and 2, which are preserved.

```
andMask = 6
orMask = 0
maskWrite(m,20,andMask,orMask)
```

Use the Modbus Explorer App

You can read and write to coils and registers using the **Modbus Explorer** app. The app supports a subset of the MATLAB Modbus functionality. You can do the following in the **Modbus Explorer**:

- Read coils, inputs, registers, and holding registers. This is the functionality of the Modbus `read` function.
- Write to coils and holding registers. This is the functionality of the Modbus `write` function.

The app does not support the functionality of the Modbus `writeRead` function or the `maskWrite` function.

The Modbus Explorer offers a graphical user interface to easily set up reads and writes, and a live plot to see the values. The read table allows you to easily organize and manage reads for multiple addresses, such as different sensors on a PLC.

To launch the **Modbus Explorer**, do one of these:

- In the MATLAB Apps tab, under **Test & Measurement**, select **Modbus Explorer**.
- At the MATLAB command line, type `modbusExplorer`.

To use the **Modbus Explorer**, you need to configure your device and connect over TCP/IP or Serial RTU. For information about how to configure and connect to your device, see “Configure a Connection in the Modbus Explorer” on page 18-20. Once a device is configured and recognized by MATLAB, it appears for selection on the **Modbus Explorer** startup screen.

After you have successfully configured your device in the **Configure** tab, click **Confirm Parameters** to open the read and write section of the **Modbus Explorer**. You can then perform reads and writes. For information about doing reads, see “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23. For information about doing writes, see “Write to Coils and Holding Registers in the Modbus Explorer” on page 18-26.

See Also

Related Examples

- “Configure a Connection in the Modbus Explorer” on page 18-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 18-26
- “Control a PLC Using the Modbus Explorer” on page 18-28
- “Generate a Script from Your Modbus Explorer Session” on page 18-33

Configure a Connection in the Modbus Explorer

The first step in using the **Modbus Explorer** to communicate with a PLC or other Modbus device is to configure the communication with the device, either over TCP/IP or Serial RTU.

Communicate over TCP/IP

1. Open the **Modbus Explorer**. On the MATLAB Apps tab, under **Test & Measurement**, select **Modbus Explorer**.
2. Choose your communication interface in the Modbus Explorer by clicking **Configure Modbus TCP/IP**.
3. On the **Configure** tab, configure the connection to your device by setting the following TCP/IP communication parameters in the toolstrip:

   Device Address: IP address of Modbus server, for example 192.168.2.20. This parameter is required to make the connection.

   Port: Remote port used by the Modbus server. The default is 502. Change it if using a different port number.

   Timeout: Maximum time in seconds to wait for a response from the Modbus server, specified as a positive value. The default is 3. You can edit the value to increase or decrease the timeout. Note that the default when using the `Timeout` property programmatically is 10 seconds. If your device requires more than the default of 3 seconds in the app, increase the value.

   Byte Order: Byte order of values written to or read from 16-bit registers. The default is Big Endian, as specified by the Modbus standard. If your device requires Little Endian, change the value in the drop-down.

   Word Order: Word order for register reads and writes that span multiple 16-bit registers. The default is Big Endian, and it is device-specific. If your device requires Little Endian, change the value in the drop-down.
4. Configure the reading of data from your device by setting the following read parameters in the toolstrip:

   Server ID: Address of the server to send the read command to. If you do not specify a Server ID, the default of 1 is used. Valid values are 1-247.

   Register Type: Target type to read. You can perform a Modbus read operation on four types of targets: coils, inputs, input registers, and holding registers.

   Register Address: Starting address to read from, specified as a double. Enter the number for your starting address.

   Precision: Data format of the register being read from on the Modbus server. For coils and inputs, the precision is always `bit`. For holding registers and input resisters, you can specify precisions such as `uint16`.
5. To test the configuration, click **Read**. If your configuration parameters are correct, the read is successful and the **Read Value** populates with the value from the read. If you see `'ERROR'` in the **Read Value** field, adjust the parameters until the read is successful.

   This value needs to match the value listed in your device manual. Make sure this value and the other configuration parameters match the specifications for your device.
6. Once you have a correct read value, click **Confirm Parameters**. The rest of the tab appears, and your device is listed in the **Device List** on the left side of the app.
7. The register details you enter in the **Configure** tab are shown in the first row of the register table. You then use the table to set up reads from your device, or press **Import** to import a table of information that you previously exported.

For information about setting up reads, see “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23.

Communicate over Serial RTU

1 Open the **Modbus Explorer**. In the MATLAB Apps tab, under **Test & Measurement**select, **Modbus Explorer**.

2 Choose your communication interface in the Modbus Explorer by clicking **Configure Modbus Serial**.

3 On **Configure** tab, configure the connection to your device by setting the following Serial RTU communication parameters in the toolstrip:

   Port: Serial port Modbus server is connected to, for example COM1.
   Baud Rate: Bit transmission rate for serial port communication. The default is 9600 bits per seconds, but the actual required value is device-dependent. Change the value in the drop-down if your device requires a different baud rate. Enter your baud rate value if it is not in the list.
   Parity: Type of parity checking. Valid choices are none (default), even, and odd. The actual required value is device-dependent. If set to the default of none, parity checking is not performed, and the parity bit is not transmitted.
   Stop Bits: Number of bits used to indicate the end of data transmission. Valid choices are 1 (default) and 2. The required value is device-dependent, though 1 is typical for even/odd parity and 2 for no parity.
   Data Bits: Number of data bits to transmit. The default is 8, which is the Modbus standard for Serial RTU. Other valid values are 5, 6, and 7.
   Timeout: Maximum time in seconds to wait for a response from the Modbus server, specified as a positive value. The default is 3. You can edit the value to increase or decrease the timeout. Note that the default when using the `Timeout` property programmatically is 10 seconds. If your device requires more than the default of 3 seconds in the app, increase the value.
   Byte Order: Byte order of values written to or read from 16-bit registers. The default is Big Endian, as specified by the Modbus standard. If your device requires Little Endian, change the value in the drop-down.
   Word Order: Word order for register reads and writes that span multiple 16-bit registers. The default is Big Endian, and it is device-specific. If your device requires Little Endian, change the value in the drop-down.

4 Configure the reading of data from your device by setting the following read parameters in the toolstrip:

   Server ID: Address of the server to send the read command to. If you do not specify a Server ID, the default of 1 is used. Valid values are 1-247.
   Register Type: Target type to read. You can perform a Modbus read operation on four types of targets: coils, inputs, input registers, and holding registers. Use the drop-down to select your type.
   Register Address: Starting address to read from, specified as a double. Enter the number for your starting address.
   Precision: Data format of the register being read from on the Modbus server. For coils and inputs, the precision is always `bit`. For holding registers and input resisters, you can specify precisions such as `uint16`.

5 To test the configuration, click **Read**. If your configuration parameters are correct, the read is successful and the **Read Value** fills in with the value from the read. If you see `'ERROR'` in the **Read Value** field, adjust the parameters until the read is successful.

This value needs to match the value listed in your device manual. Make sure this value and the other configuration parameters match the specifications for your device.

6 Once you have a correct read value, click **Confirm Parameters**. The rest of the tab appears, and your device is listed in the **Device List** on the left side of the app.

7 The register details you enter in the **Configure** tab are shown in the first row of the register table. You then use the table to set up reads from your device, or press **Import** to import a table of information that you previously exported.

   For information about setting up reads, see “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23.

See Also

Related Examples

- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 18-26
- “Control a PLC Using the Modbus Explorer” on page 18-28
- “Generate a Script from Your Modbus Explorer Session” on page 18-33

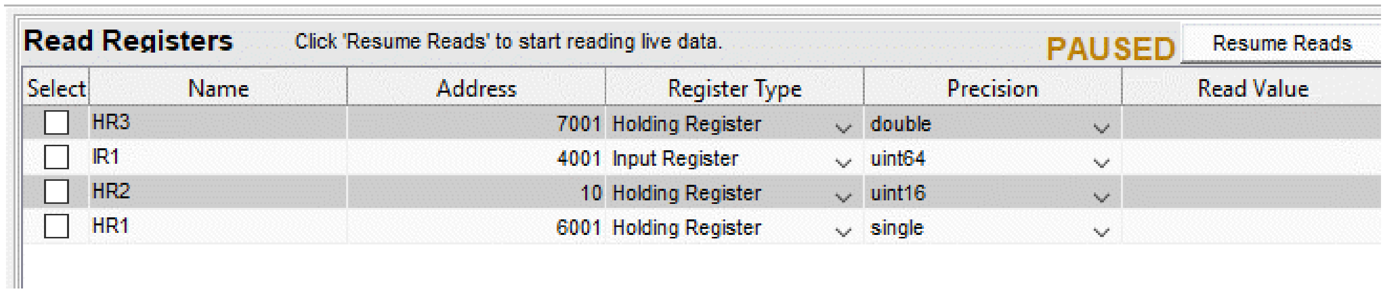
Read Coils, Inputs, and Registers in the Modbus Explorer

You can read coils, inputs, input registers, and holding registers in the **Modbus Explorer**. This is the functionality of the Modbus `read` function.

You must first configure your device before performing read operations. For information on how to configure and connect to your device, see “Configure a Connection in the Modbus Explorer” on page 18-20.

1. To perform a read operation, enter the information about the coils, inputs, registers, or holding registers you want to control in the **Read Registers** table. The first row of the table is already filled in with the register you configured on the **Configure** tab.
2. To set up another register, click **Insert**. For each inserted row, click the **Address** field and enter the address of the coil, input, input register, or holding register you want to read values from.
3. In the **Register Type** column, click the down arrow to select the target type to read. You can perform a Modbus read operation on four types of targets: coils, inputs, input registers, and holding registers.
4. In the **Precision** column, click the down arrow to select the precision. Choose the data format of the register being read from on the Modbus server.
5. In the **Name** column, enter a name. A default name is provided, and you can keep it or change it.
6. Once you have entered all of the fields, click **Resume Reads** to start reading.

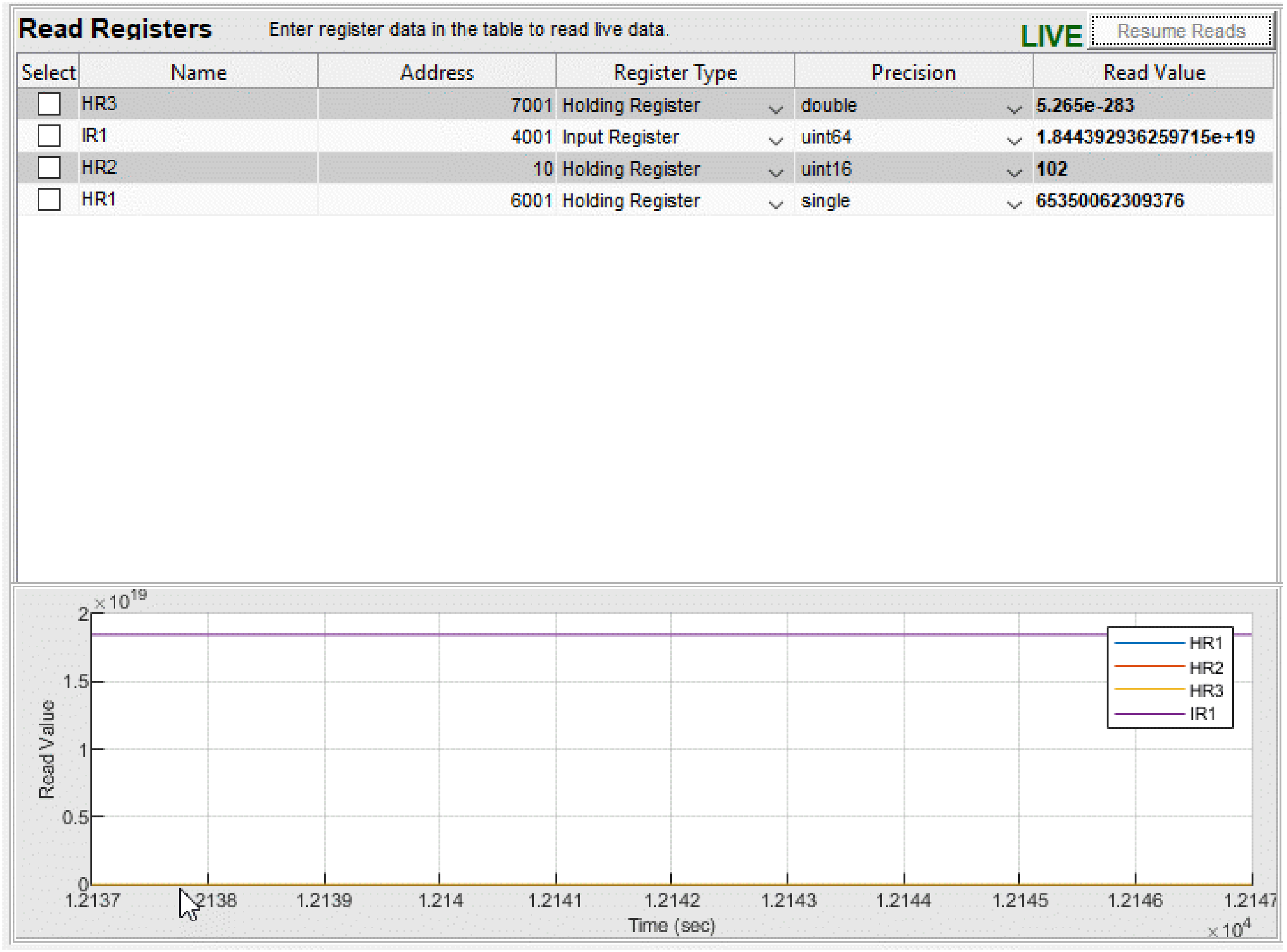
   If the read is successful, the **Read Value** column fills in with the read value. If you see `'ERROR'` in the **Read Value** field, adjust the parameters until the read is successful.
7. To read multiple registers, add rows to the table by clicking **Insert** on the toolstrip. New rows are added to the top of the table. You can add as many rows as you need. For each row, fill in all of the fields for that read. While you are editing a row, reading is paused, as indicated by the `PAUSED` status to the left of the **Resume Reads** button.



Select	Name	Address	Register Type	Precision	Read Value
☐	HR3	7001	Holding Register	double	
☐	IR1	4001	Input Register	uint64	
☐	HR2	10	Holding Register	uint16	
☐	HR1	6001	Holding Register	single	

8. After you set up the table, click **Resume Reads**.

   The reads are performed and the status changes to `LIVE`. You can also see the read values in the live plot.



Read Registers Enter register data in the table to read live data.

Select	Name	Address	Register Type	Precision	Read Value
☐	HR3	7001	Holding Register	double	5.265e-283
☐	IR1	4001	Input Register	uint64	1.844392936259715e+19
☐	HR2	10	Holding Register	uint16	102
☐	HR1	6001	Holding Register	single	65350062309376

9 If you want to alter the plot, use the **Plot Tools** section. You can change the axes, show or hide the legend, and select which registers to display in the plot. The plot changes dynamically when you change any of these factors using the plot tools.

Edit the Read Registers Table

You can manipulate the register table by using the buttons in the toolstrip and the check boxes in the table.

- **Insert** - Insert a blank row at the top of the table.
- **Delete** - Delete selected rows. Use the check boxes in the table to select rows to delete.
- **Move Up** - Move up selected rows one position in the table. Use the check boxes in the table to select rows to move.
- **Move Down** - Move down selected rows one position in the table. Use the check boxes in the table to select rows to move.
- **Sort** - Sort the register table data by column value. Click **Sort** and choose **Name** to sort in alphabetic order of name, **Address** for ascending order of address, **Register Type** for alphabetical order of register type, or **Precision** for alphabetical order of precision.

Import or Export Read Data

You can export the contents of the **Read Registers** table to use again later.

To export the register table configuration, click the **Export** button in the toolstrip. In the Save dialog box, choose a file name and navigate to the save location, then click **Save**. The table is saved as a MAT-file.

To import the contents back into the **Read Registers** table, click **Import** in the toolstrip. In the Load dialog box, navigate to the saved MAT-file, select it, and click **Open**.

See Also

Related Examples

- “Configure a Connection in the Modbus Explorer” on page 18-20
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 18-26
- “Control a PLC Using the Modbus Explorer” on page 18-28
- “Generate a Script from Your Modbus Explorer Session” on page 18-33

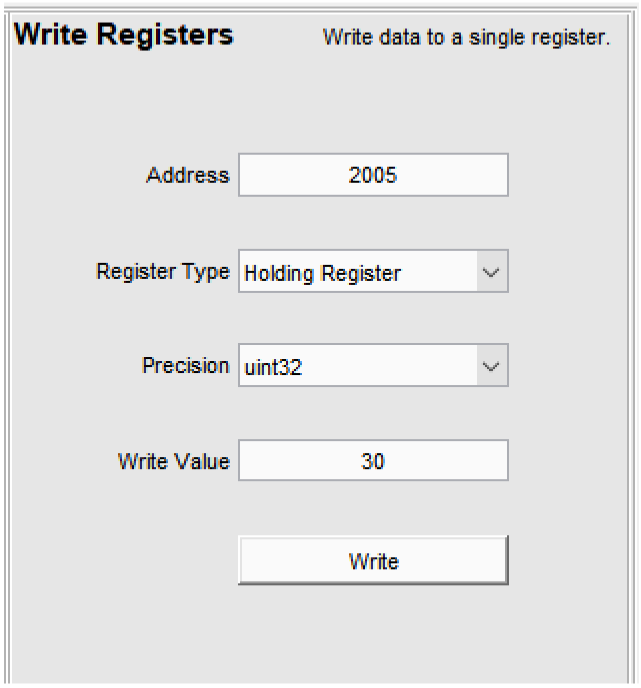
Write to Coils and Holding Registers in the Modbus Explorer

You can write to coils and holding registers in the **Modbus Explorer**. This is the functionality of the Modbus `write` function.

You must first configure your device before performing write or read operations. For information on how to configure and connect to your device, see “Configure a Connection in the Modbus Explorer” on page 18-20. For more information on performing reads, see “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23.

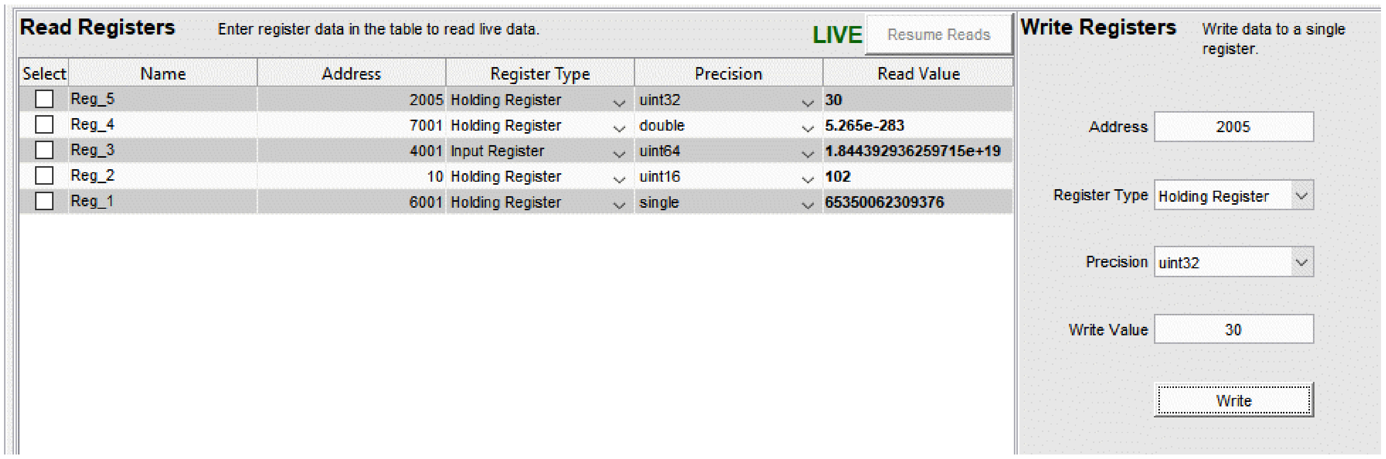
Use the **Write Registers** section of the app to perform write operations.

1. Enter the information about the register you want to control in the **Write Registers** section. You can do one write at a time using this section of the app.
2. To set up the write, click the **Address** field and enter the address of the register you want to write a value to.
3. In the **Register Type** field, click the down arrow to select the register type of the address. You can perform a Modbus write operation on two types of targets: coils and holding registers.
4. In the **Precision** field, click the down arrow to select the precision. This is the data format of the register being written to on the Modbus server.
5. In the **Write Value** field, enter the value to write. For coils and inputs, you can only write values of `0` or `1`. For input registers and holding registers you can write other values.
6. Once you have entered all the fields, the **Write** button becomes activated.



Address: 2005
Register Type: Holding Register
Precision: uint32
Write Value: 30

7. To send the value to the register, click **Write**.
8. If you have the same register listed in the **Read Registers** table, you see the read value update when you click **Write**. In the example shown here, you can see that the value of 30 was sent to the register and that it is now reflected in the read table for `Reg_5` in the first row.


Read Registers
Enter register data in the table to read live data.
LIVE
Resume Reads
Write Registers
Write data to a single register.
Write

Select	Name	Address	Register Type	Precision	Read Value
☐	Reg_5	2005	Holding Register	uint32	30
☐	Reg_4	7001	Holding Register	double	5.265e-283
☐	Reg_3	4001	Input Register	uint64	1.844392936259715e+19
☐	Reg_2	10	Holding Register	uint16	102
☐	Reg_1	6001	Holding Register	single	65350062309376

Address 2005

Register Type Holding Register

Precision uint32

Write Value 30

See Also

Related Examples

- “Configure a Connection in the Modbus Explorer” on page 18-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23
- “Control a PLC Using the Modbus Explorer” on page 18-28
- “Generate a Script from Your Modbus Explorer Session” on page 18-33

Control a PLC Using the Modbus Explorer

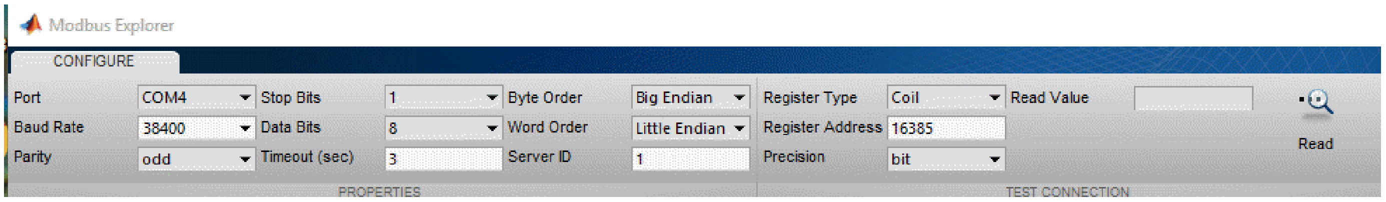
This example shows how to perform read and write operations to a PLC using the **Modbus Explorer**. The PLC is a Click Koyo cube with registers that can be used in industrial control and other industrial applications including controlling switches, timers, and sensors.

1. Open the **Modbus Explorer**. In the MATLAB Apps tab, under **Test & Measurement**, select **Modbus Explorer**.
2. The device is accessed over Serial RTU. To choose the communication interface in the Modbus Explorer click **Device** then **Modbus Serial** in the toolstrip.
3. On the **Configure** tab, configure the connection to your device by setting the following Serial RTU communication parameters in the toolstrip:

   Port: Serial port Modbus server is connected to. Set to `COM4`.
   Baud Rate: Bit transmission rate for serial port communication. The default is 9600 bits per seconds. Change it to 38400.
   Parity: Type of parity checking. Valid choices are none (default), even, and odd, and the actual required value is device-dependent. Set it to odd.
   Stop Bits: Number of bits used to indicate the end of data transmission. Valid choices are 1 (default) and 2, and the actual required value is device-dependent. Keep the default.
   Data Bits: Number of data bits to transmit. The default is 8, which is the Modbus standard for Serial RTU. Other valid values are 5, 6, and 7. Keep the default.
   Timeout: Maximum time in seconds to wait for a response from the Modbus server. The default is 3. You can edit the value to increase or decrease the timeout. Keep the default.
   Byte Order: Byte order of values written to or read from 16-bit registers. The default is Big Endian, as specified by the Modbus standard. Keep the default.
   Word Order: Word order for register reads and writes that span multiple 16-bit registers. The default is Big Endian, and it is device-specific. Set it to Little Endian.
4. Configure the reading of data from your device by setting the following read parameters in the toolstrip:

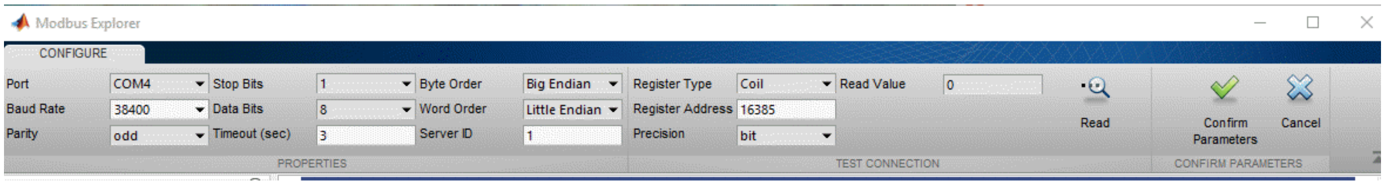
   Server ID: Address of the server to send the read command to, specified as a double. Valid values are 0-247, with 0 being the broadcast address. Set to 1.
   Register Type: Target area to read. You can perform a Modbus read operation on four types of targets: coils, inputs, input registers, and holding registers. Use the drop-down to select Coil.
   Register Address: Starting address to read from, specified as a double. Enter the number for your starting address, `16385` in this case.
   Precision: Data format of the register being read from on the Modbus server. For coils and inputs, the precision is always `bit`. For holding registers and input resisters, you can specify precisions such as `uint16`.

   The configuration should look like this after you configure the communication and read settings.



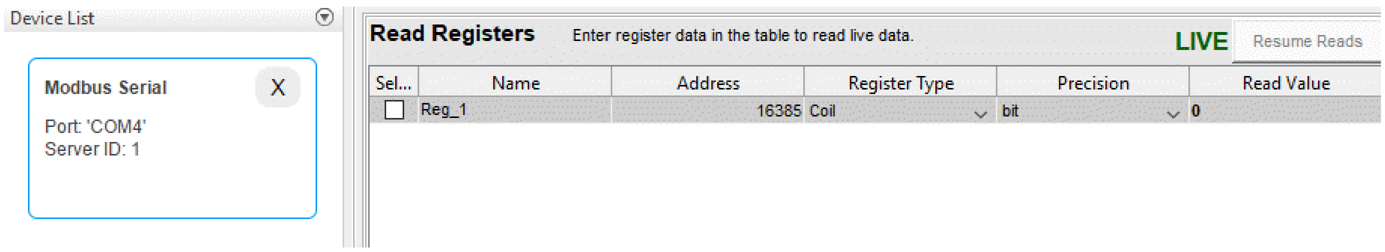
Port	COM4	Stop Bits	1	Byte Order	Big Endian	Register Type	Coil	Read Value	
Baud Rate	38400	Data Bits	8	Word Order	Little Endian	Register Address	16385		
Parity	odd	Timeout (sec)	3	Server ID	1	Precision	bit		

5. To test the configuration, click **Read**. If your configuration parameters are correct, the read is successful and the **Read Value** populates with the value from the read operation. If you get an error, adjust the parameters until the read is successful. In this case, the value should be 0.



Port	COM4	Stop Bits	1	Byte Order	Big Endian	Register Type	Coil	Read Value	0
Baud Rate	38400	Data Bits	8	Word Order	Little Endian	Register Address	16385		
Parity	odd	Timeout (sec)	3	Server ID	1	Precision	bit		

6 After you have a correct read value, click **Confirm Parameters**. The **Configure** tab disappears and the **Modbus Explorer** tab appears, and your device is listed in the **Device List** on the left side of the app, as shown here.



Port: 'COM4'
Server ID: 1

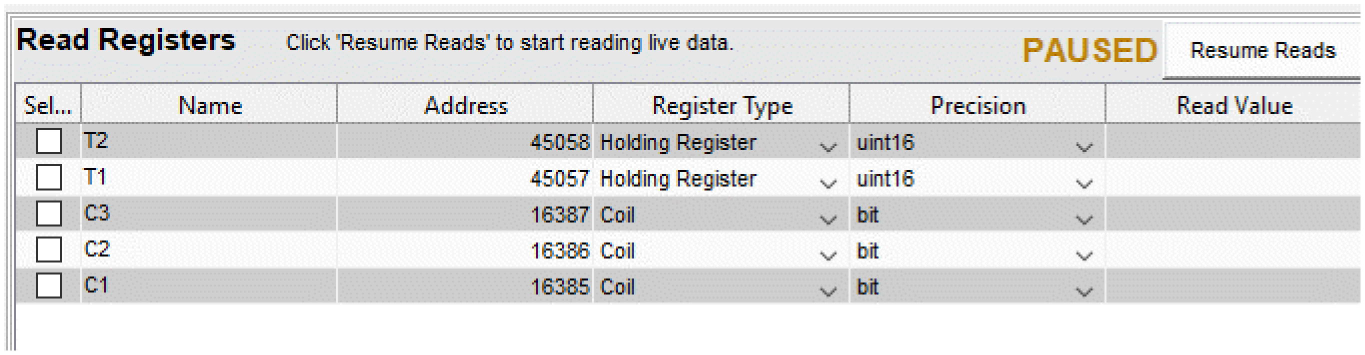
Read Registers Enter register data in the table to read live data.

Sel...	Name	Address	Register Type	Precision	Read Value
☐	Reg_1	16385	Coil	bit	0

7 You then use the table to set up more reads from your device. Fill in the **Read Registers** table to read data from two timers and three switches. Since the table automatically displays the register you configure in the **Configure** tab, the first timer is already listed. Change the name to `C1`, then add four more rows so you have these reads set up.

```
Switches
C1, Address 16385, Coil, bit
C2, Address 16386, Coil, bit
C3, Address 16387, Coil, bit
Timers
T1, Address 45057, Holding Register, uint16
T2, Address 45058, Holding Register, uint16
```

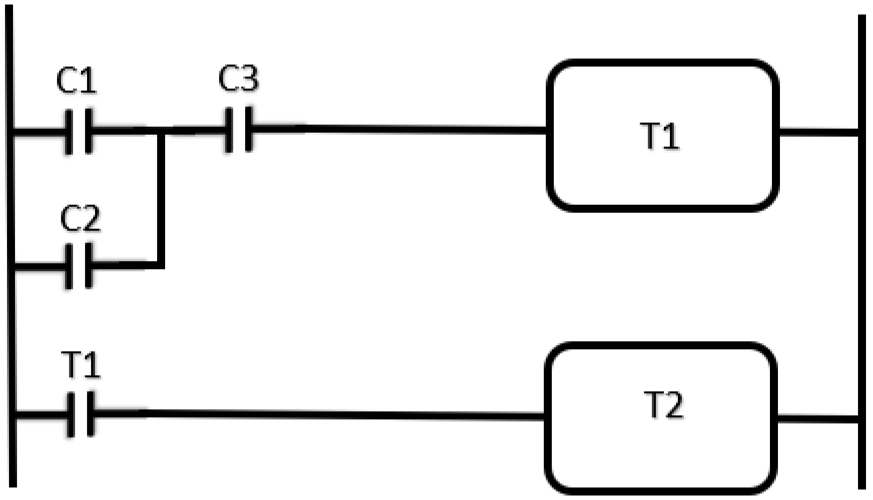
The table should look like this:



Read Registers Click 'Resume Reads' to start reading live data.

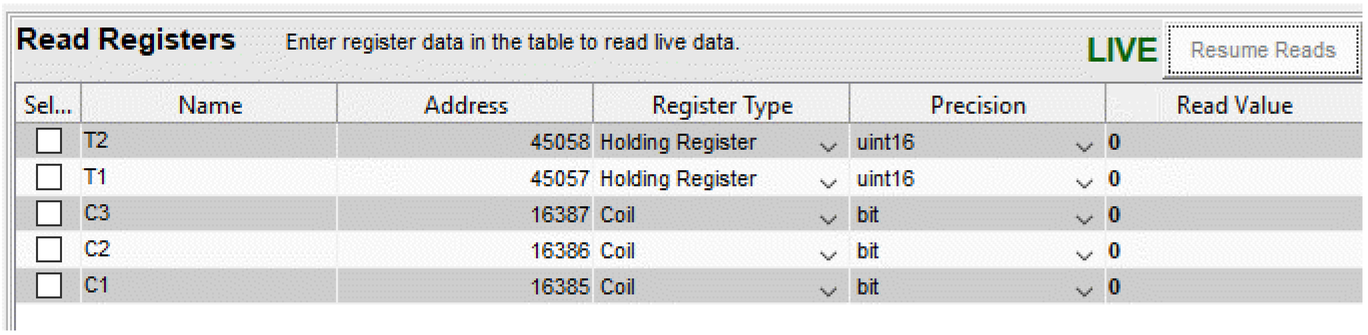
Sel...	Name	Address	Register Type	Precision	Read Value
☐	T2	45058	Holding Register	uint16	
☐	T1	45057	Holding Register	uint16	
☐	C3	16387	Coil	bit	
☐	C2	16386	Coil	bit	
☐	C1	16385	Coil	bit	

The PLC that contains these timers and switches is shown here.



8 To perform the reads on the five registers in the table, click **Resume Reads**.

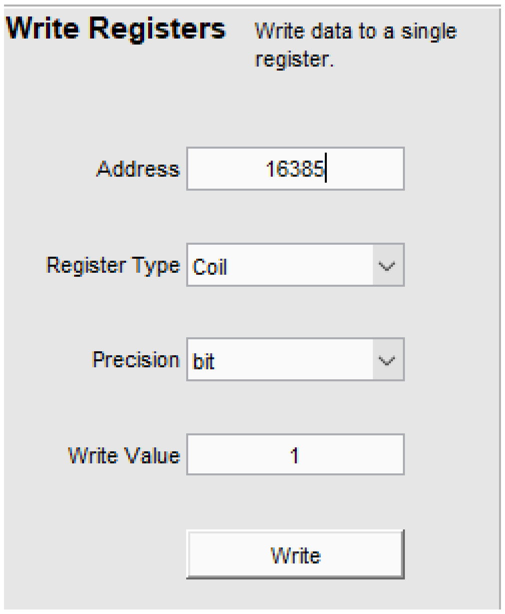
The **Read Value** column displays the value that is returned and the status indicator changes to `LIVE`, as shown here.


Read Registers Enter register data in the table to read live data. LIVE Resume Reads

Sel...	Name	Address	Register Type	Precision	Read Value
☐	T2	45058	Holding Register	uint16	0
☐	T1	45057	Holding Register	uint16	0
☐	C3	16387	Coil	bit	0
☐	C2	16386	Coil	bit	0
☐	C1	16385	Coil	bit	0

In this case, the value of 0 means the switch or timer is connected and available, but it is not activated.

9 To turn on one of the switches, `C1`, perform a write to the register. In the **Write Registers** section, fill in the following:



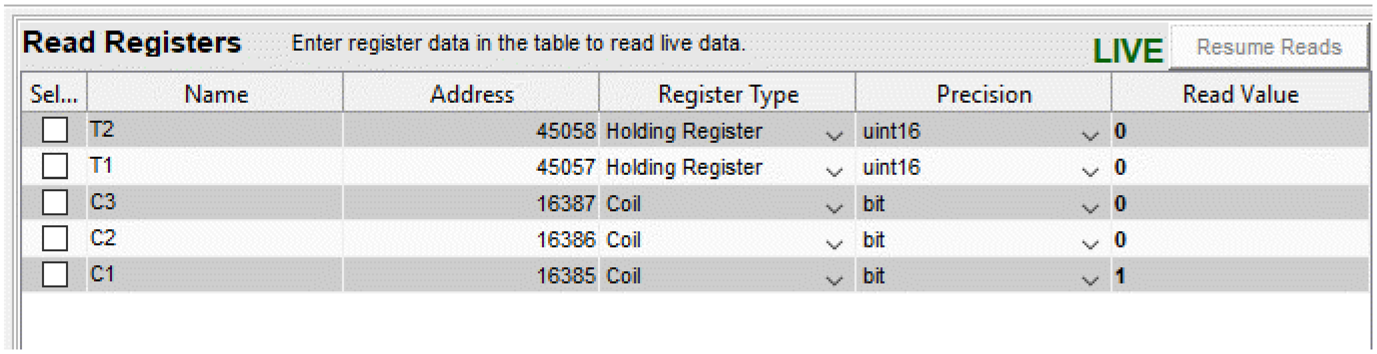
Write Registers Write data to a single register.

Address	16385
Register Type	Coil
Precision	bit
Write Value	1

After you have entered all of the fields, the **Write** button becomes activated.

10 To send the value to the register, click **Write**.

Since you have the same register listed in the **Read Registers** table, you see the read value update when you click **Write**. In the example shown here, you can see that the value of 1 was sent to the register and that it is now reflected in the read table for C1, indicating that the switch is turned on.



Read Registers Enter register data in the table to read live data.

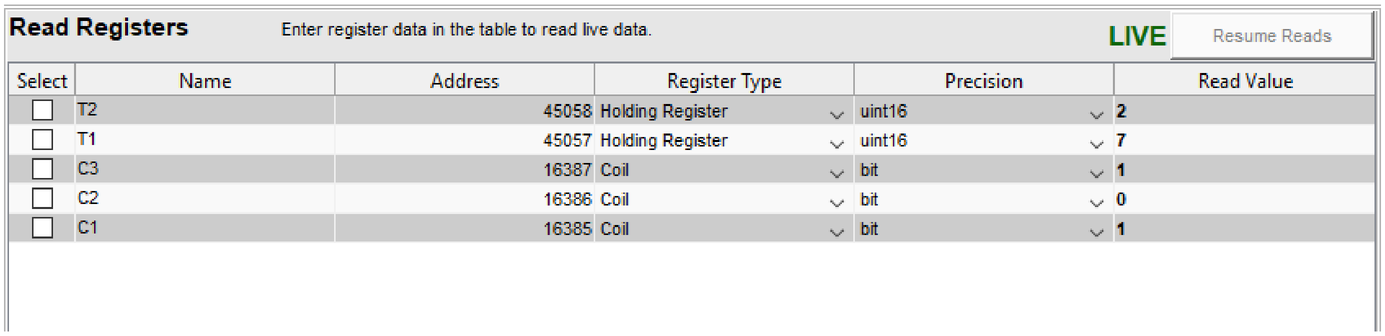
Sel...	Name	Address	Register Type	Precision	Read Value
☐	T2	45058	Holding Register	uint16	0
☐	T1	45057	Holding Register	uint16	0
☐	C3	16387	Coil	bit	0
☐	C2	16386	Coil	bit	0
☐	C1	16385	Coil	bit	1

11 Perform another write to turn the C3 switch on. In the **Write Registers** section, fill in the following:

```
Address: 16387
Type: Coil
Precision: bit
Write Value: 1
```

Click **Write**.

Once that switch is on, the timers turn on, since that is how the PLC board is arranged. T1 is turned on when the switches are on, and then 5 seconds later T2 is automatically turned on. At that point both of the timers and two of the switches are turned on, as shown here.



Read Registers Enter register data in the table to read live data. LIVE Resume Reads

Select	Name	Address	Register Type	Precision	Read Value
☐	T2	45058	Holding Register	uint16	2
☐	T1	45057	Holding Register	uint16	7
☐	C3	16387	Coil	bit	1
☐	C2	16386	Coil	bit	0
☐	C1	16385	Coil	bit	1

See Also

Related Examples

- “Configure a Connection in the Modbus Explorer” on page 18-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 18-26
- “Generate a Script from Your Modbus Explorer Session” on page 18-33

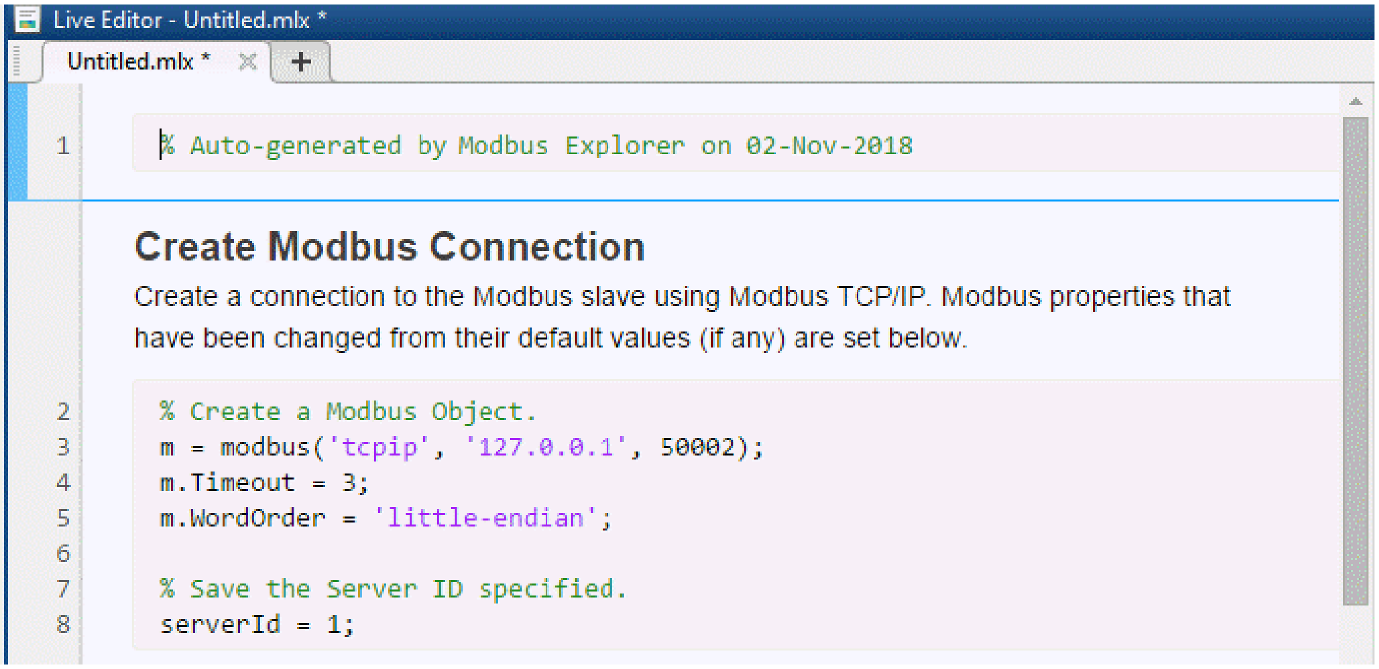
Generate a Script from Your Modbus Explorer Session

You can generate a MATLAB script from your **Modbus Explorer** session, and then run it at the command line using the toolbox Modbus functionality. The generated script contains your device configuration, all the read operations that you perform, the last write operation that you perform for each register type, and cleanup tasks.

Note that generating a script is not the same as saving the contents of the **Read Register** table. To do that, use **Export** as described in “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23.

To generate a script from your Modbus Explorer session, click **Generate Script** in the toolstrip. The script appears in the MATLAB Editor as a live script. To keep the generated script, save it in the editor. An example of a script is shown here.

Create Modbus Connection - This section of the generated script creates the `modbus` object using the configuration properties specified.



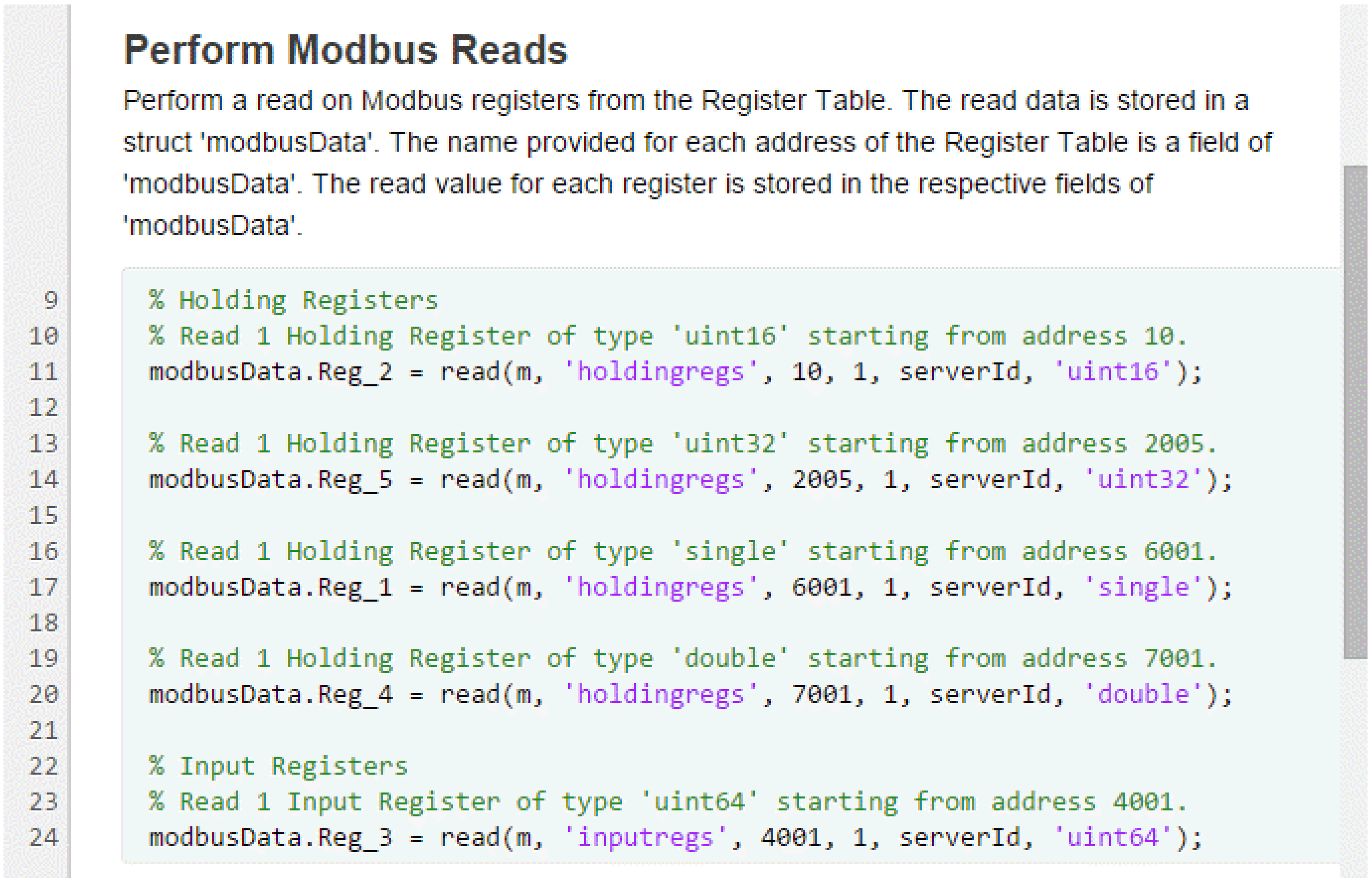
```
% Auto-generated by Modbus Explorer on 02-Nov-2018
```

Create Modbus Connection

Create a connection to the Modbus slave using Modbus TCP/IP. Modbus properties that have been changed from their default values (if any) are set below.

```
% Create a Modbus Object.
m = modbus('tcpip', '127.0.0.1', 50002);
m.Timeout = 3;
m.WordOrder = 'little-endian';

% Save the Server ID specified.
serverId = 1;
```

Perform Modbus Reads - This section of the script performs all the Modbus reads that were done in your session. In this example, five reads were performed.

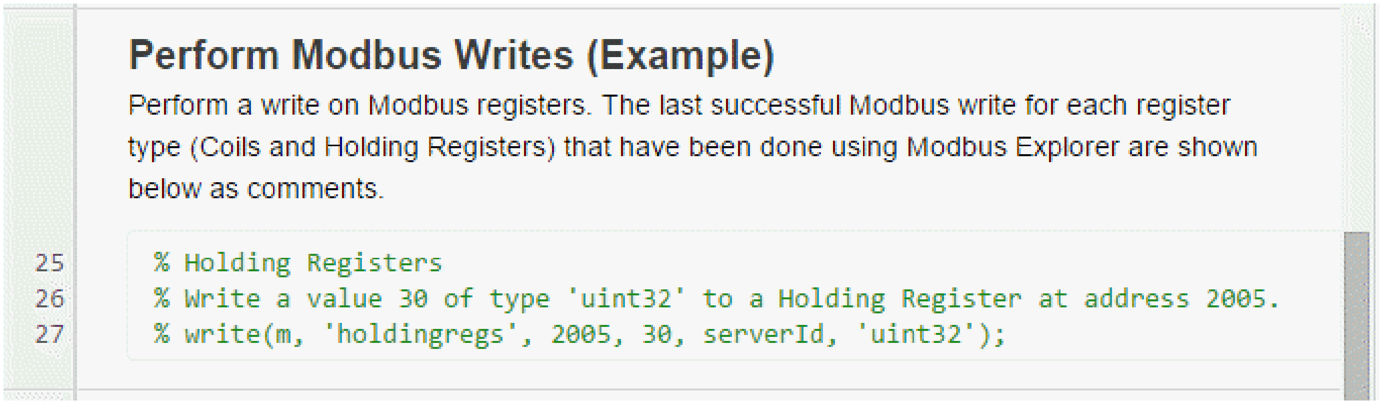


Perform Modbus Reads

Perform a read on Modbus registers from the Register Table. The read data is stored in a struct 'modbusData'. The name provided for each address of the Register Table is a field of 'modbusData'. The read value for each register is stored in the respective fields of 'modbusData'.

```
% Holding Registers
% Read 1 Holding Register of type 'uint16' starting from address 10.
modbusData.Reg_2 = read(m, 'holdingregs', 10, 1, serverId, 'uint16');

% Read 1 Holding Register of type 'uint32' starting from address 2005.
modbusData.Reg_5 = read(m, 'holdingregs', 2005, 1, serverId, 'uint32');

% Read 1 Holding Register of type 'single' starting from address 6001.
modbusData.Reg_1 = read(m, 'holdingregs', 6001, 1, serverId, 'single');

% Read 1 Holding Register of type 'double' starting from address 7001.
modbusData.Reg_4 = read(m, 'holdingregs', 7001, 1, serverId, 'double');

% Input Registers
% Read 1 Input Register of type 'uint64' starting from address 4001.
modbusData.Reg_3 = read(m, 'inputregs', 4001, 1, serverId, 'uint64');
```

Perform Modbus Writes - This section of the script shows the last Modbus write operation that was done in the session for each register type. It is presented as a comment so that you must intentionally uncomment it to do the write, to prevent unexpected actions on your device.

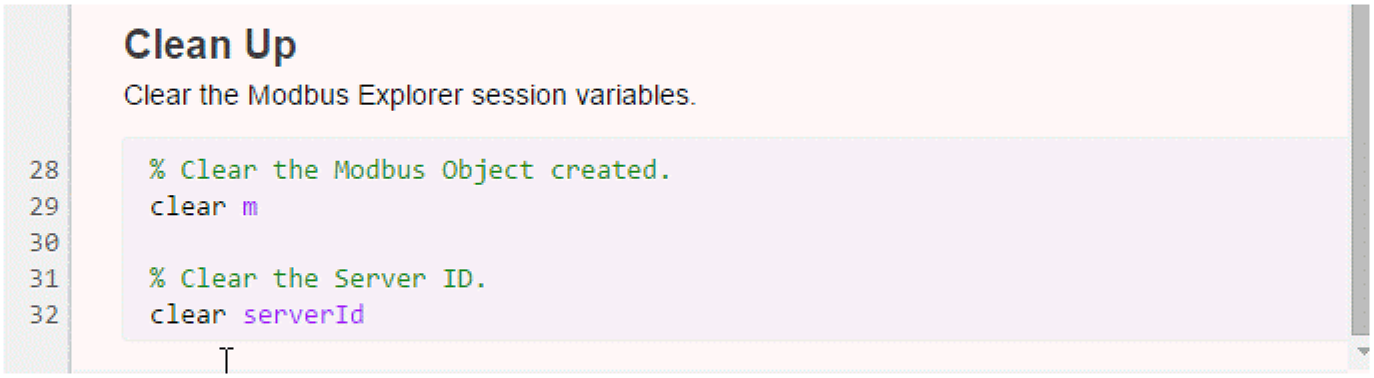


Perform Modbus Writes (Example)

Perform a write on Modbus registers. The last successful Modbus write for each register type (Coils and Holding Registers) that have been done using Modbus Explorer are shown below as comments.

```
% Holding Registers
% Write a value 30 of type 'uint32' to a Holding Register at address 2005.
% write(m, 'holdingregs', 2005, 30, serverId, 'uint32');
```

Clean Up - This section of the script clears the `modbus` object and releases the server ID.



Clean Up

Clear the Modbus Explorer session variables.

```
% Clear the Modbus Object created.
clear m

% Clear the Server ID.
clear serverId
```

See Also

Related Examples

- “Configure a Connection in the Modbus Explorer” on page 18-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 18-26
- “Control a PLC Using the Modbus Explorer” on page 18-28

Troubleshooting the Modbus Interface

In this section...

"Supported Platforms" on page 18-36
"Configuration and Connection" on page 18-36
"Other Troubleshooting Tips for Modbus" on page 18-37

Industrial Communication Toolbox supports the Modbus interface over TCP/IP or Serial RTU. You can use it to communicate with Modbus servers, such as controlling a PLC (Programmable Logic Controller), communicating with a temperature controller, controlling a stepper motor, sending data to a DSP, reading bulk memory from a PAC controller, or monitoring temperature and humidly on a Modbus probe.

Using the Modbus interface, you can do the following tasks:

- Read coils, inputs, input registers, and holding registers
- Write to coils and holding registers
- Perform a combination of one write operation and one read operation on groups of holding registers in a single Modbus transaction
- Modify the contents of a holding register using a mask write operation

Supported Platforms

Industrial Communication Toolbox supports the Modbus interface over TCP/IP or Serial RTU. It is supported on the following platforms.

- Linux 64-bit
- Mac OS 64-bit
- Microsoft Windows 64-bit

Configuration and Connection

1. If you are connecting to a local or remote device over Modbus, make sure that the device is powered on and available.
2. Industrial Communication Toolbox can communicate over Modbus using TCP/IP or Serial RTU. If you are connecting via TCP/IP, you need to know the IP address or host name of the Modbus server. If you are connecting via Serial RTU, you need to specify the Serial port the Modbus server is connected to.
3. Make sure you can create the `modbus` object with its necessary arguments. For examples of creating the object and information about the required arguments, see "Create a Modbus Connection" on page 18-3.

   When you create the `modbus` object, it connects you to the server or device. There is no separate connection function required.
4. When you have connected, you can communicate with your device. See "Read Temperature from a Remote Temperature Sensor" on page 18-13 for an example of communicating with a device. See "Other Troubleshooting Tips for Modbus" on page 18-37 for tips about communication issues after initial connection.

Other Troubleshooting Tips for Modbus

These tips may be relevant to your use of the Modbus interface.

Address Range

When specifying read and write addresses, the addresses must be in the range 0–65535.

Underlying Interface

You might encounter connection problems that are due to the underlying TCP/IP or Serial Port connections, rather than being specific to the Modbus interface.

Modbus Addresses

If you have trouble figuring out a Modbus address, see the vendor documentation of the device. For example, you may need to map a PLC register to the Modbus address for the register. The vendor documentation may help.

Some vendors include an extra digit in addresses that gets dropped. For example 43233 is really address 3233. Devices are usually represented by a four-digit address, and some vendors use a 5th digit to represent the type of target, for example, coils. So you may need to adjust an address to account for this if your device vendor does that.

The Modbus functions use 1-based addressing, not 0-based addressing like Modbus uses. The toolbox subtracts 1 from any addresses that are passed in via the address parameters in the read and write functions.

OPC Information Reference

OPC Quality

OPC Quality

Industrial Communication Toolbox software uses specific quality attributes defined by the OPC Foundation, based on a major quality value, a substatus for that major quality value, and a limit status indicating how the value is limited. This appendix describes the standard quality attributes defined by the OPC Foundation that are used in the toolbox, and describes any special extensions that the toolbox uses.

An OPC quality value is a number ranging from 0 to 65535, made up of four parts. The high 8 bits of the quality value represent the vendor-specific quality information. The low 8 bits are arranged as `QQSSSSLL`, where `QQ` represents the major quality, `SSSS` represents the quality substatus, and `LL` represents the limit status.

OPC HDA quality values are layered on top of OPC DA quality values.

The following topics describe the OPC quality values and texts associated with each quality part.

- "Major Quality" on page A-3
- "Quality Substatus" on page A-4
- "Limit Status" on page A-6

For more information, see the Quality property reference page. The quality of an item is also stored in native value format in the QualityID property of the `daitem` object.

Major Quality

Industrial Communication Toolbox uses the following major quality values and text. The major quality is contained in bits 7 and 8 of the quality value.

Major Quality Values

Value	Quality Text	Description
0	`Bad`	The value is not useful for the reason indicated by the substatus. The table Bad Quality Substatus Values contains information about the substatus for bad quality.
1	`Uncertain`	The quality of the value is uncertain for reasons indicated by the substatus. The table Uncertain Quality Substatus Values contains information about the substatus for uncertain quality.
3	`Good`	The quality of the value is good. The table Good Quality Substatus Values contains information about the substatus for good quality.
N/A	`Repeat`	The value is repeated from a previous known value for this item. This toolbox-specific value occurs only in data returned from `getdata` or `opcread`, when you request array formatted values.

See Also

More About

- “OPC Quality” on page A-2
- “Quality Substatus” on page A-4
- “Limit Status” on page A-6

Quality Substatus

Each major quality status has an additional substatus that describes the quality of the value in more detail. The following tables describe the quality substatus for each major quality.

- Good Quality Substatus Values
- Uncertain Quality Substatus Values
- Bad Quality Substatus Values

Good Quality Substatus Values

Value	Substatus Text	Description
0	`Non-specific`	The value is good. There are no special conditions.
6	`Local Override`	The value has been overridden. Typically, this means that the device has been disconnected from the OPC server (either physically, or through software) and a manually entered value has been forced.

Uncertain Quality Substatus Values

Value	Substatus Text	Description
0	`Non-Specific`	The server has not published a specific reason why the value is uncertain.
1	`Last Usable Value`	Whatever was writing the data value has stopped doing so. The returned value should be regarded as "stale." Note that this quality value differs from `'Bad: Last Known Value'` in that the "bad" quality is associated specifically with a detectable communications error. The `'Uncertain: Last Usable Value'` text is associated with the failure of some external source to "put" something into the value within an acceptable period of time. You can examine the age of the value using the TimeStamp property associated with this quality.
4	`Sensor Not Accurate`	Either the value has pegged at one of the sensor limits, or the sensor is otherwise known to be out of calibration via some form of internal diagnostics.
5	`Engineering Units Exceeded`	The returned value is outside the limits defined for this value. Note that this substatus does not imply that the value is pegged at some upper limit. The value may exceed the engineering units even further in future updates.
6	`Sub-Normal`	The value is derived from multiple sources and has less than the required number of good sources.

Bad Quality Substatus Values

Value	Substatus Text	Description
0	`Non-Specific`	The value is bad but no specific reason is known.
1	`Configuration Error`	There is some server-specific problem with the configuration. For example, the item in question is deleted from the running server configuration.
2	`Not Connected`	The input is required to be logically connected to something, but is not connected. This quality may reflect that no value is available at this time, possibly because the data source has not yet provided one.
3	`Device Failure`	A device failure has been detected.
4	`Sensor Failure`	A sensor failure has been detected.
5	`Last Known Value`	Communication between the device and the server has failed. However, the last known value is available. Note that the age of the last known value can be determined from the TimeStamp property.
6	`Comm Failure`	Communication between the device and server has failed. There is no last known value available.
7	`Out of Service`	The Active state of the item or group containing the item is set to `off`. This quality is also used to indicate that the item is not being updated by the server for some reason.

See Also

More About

- "OPC Quality" on page A-2
- "Major Quality" on page A-3
- "Limit Status" on page A-6

Limit Status

The limit status is not dependent on the major quality and substatus parts of a quality value.

The following table lists the limit status values and texts used in Industrial Communication Toolbox.

Value	Limit Status Text	Description
0	`Not Limited`	The value is free to move. Note that when the limit status has this value, it is omitted from any quality attribute in the toolbox.
1	`Low Limited`	The value is fixed at some lower limit.
2	`High Limited`	The value is fixed at some upper limit.
3	`Constant`	The value is a constant and cannot change.

See Also

More About

- “OPC Quality” on page A-2
- “Major Quality” on page A-3
- “Quality Substatus” on page A-4

OPC DA Server Item Properties

OPC DA Server Item Properties

All server items defined in an OPC server name space have associated properties that describe that server item in more detail. The properties defined by the OPC Foundation are described in these topics:

- “OPC Item Property Set” on page B-3
- “OPC Specific Properties” on page B-4
- “OPC Recommended Properties” on page B-5

For more information on querying OPC server item properties, consult the help for `serveritemprops`.

OPC Item Property Set

Every item defined by an OPC server has specific attributes, or properties, that describe that server item in more detail. These properties include the current Value, Quality and TimeStamp for the server item, plus additional properties that a server may require in order to determine the quality of a value, or to decide whether to generate a `DataChange` event for groups that have a nonzero DeadbandPercent value. Exposure of the server item properties to a client is intended to provide a client with more information on a specific item, and is not intended to provide efficient access to large amounts of data. Rather, you should use the `read` function to read data from a large number of server items.

Each property is identified by a Property ID, or *PropID*, which is an integer value. The OPC Data Access Specification defines three sets of these properties, based on their PropID.

OPC Item Property Sets

Set Name	ID Range	Description
OPC Specific	1-99	Information directly related to the OPC server for that item.
OPC Recommended	100-4999	Additional information which is commonly associated with items, such as ranges of valid values, alarm limits, etc.
Vendor Specific	5000 or greater	Specific properties defined by an OPC server vendor. Since these vary from vendor to vendor, the actual descriptions are not presented in this appendix.

Each of the property sets defined by the OPC Foundation is presented in the following sections.

Note OPC servers must implement the OPC specific properties. However, the recommended properties are not mandatory, and an OPC server could provide any subset of the recommended properties, or none of them.

See Also

More About

- “OPC DA Server Item Properties” on page B-2
- “OPC Specific Properties” on page B-4
- “OPC Recommended Properties” on page B-5

OPC Specific Properties

OPC Specific Properties

PropID	Description
1	“Item Canonical DataType” The data type of the item as stored on the OPC server. This property is also exposed in the CanonicalDataType property of the daitem object.
2	“Item Value” The value that was last obtained from the OPC server for the item. This property is the same as the Value property of the `daitem` object. Querying this property behaves like a `read` operation from the device.
3	“Item Quality” The quality of the item's `Value` property. This property is the same as the Quality property of the `daitem` object. Querying this property behaves like a `read` operation from the device.
4	“Item Timestamp” The time that the Value and Quality was obtained by the device (if this is available) or the time the server updated or validated the `Value` and `Quality` in its cache. This property is the same as the TimeStamp property of the `daitem` object. Querying this property behaves like a `read` operation from the device.
5	“Item Access Rights” The ability of the server to read or write data to this item.
6	“Server Scan Rate” Represents the fastest rate at which the server could obtain data from the underlying data source. The accuracy of this value could be affected by system load and other factors, and is not a guaranteed rate.
7-99	Reserved for future use

See Also

More About

- “OPC DA Server Item Properties” on page B-2
- “OPC Item Property Set” on page B-3
- “OPC Recommended Properties” on page B-5

OPC Recommended Properties

The Recommended Properties are divided into the following tables.

- Recommended Properties Related to the Item Value
- Recommended Properties Related to Operator Displays
- Recommended Properties Related to Alarm and Condition Values

Recommended Properties Related to the Item Value

PropID	Description
100	“EU Units” The engineering units for this item.
101	“Item Description” A description of the item.
102	“High EU” Present only for analog data. Represents the highest value likely to be obtained in normal operation. Also used by servers that support non-zero DeadbandPercent values for a group.
103	“Low EU” Present only for analog data. Represents the lowest value likely to be obtained in normal operation. Also used by servers that support non-zero DeadbandPercent values for a group.
104	“High Instrument Range” Represents the highest value that can be returned by the instrument.
105	“Low Instrument Range” Represents the highest value that can be returned by the instrument.
106	“Contact Close Label” Present only for discrete data. Represents text to be associated with this contact when it is in the closed (non-zero) state.
107	“Contact Open Label” Present only for discrete data. Represents text to be associated with this contact when it is in the open (zero) state.
108	“Item Timezone” The difference in minutes between the item’s UTC Timestamp and the local time in which the item value was obtained. Industrial Communication Toolbox software does not use this property to adjust time stamps for an item.
109-199	Reserved for future use.

Recommended Properties Related to Operator Displays

PropID	Description
200	"Default Display" The name of an operator display associated with this item.
201	"Current Foreground Color" The COLORREF in which the item should be displayed.
202	"Current Background Color" The COLORREF in which the item should be displayed.
203	"Current Blink" Defines whether a display of this item should blink.
204	"BMP File" Bitmap file associated with this item.
205	"Sound File" .WAV or .MID file associated with this item.
206	"HTML File" URL reference for this item.
207	"AVI File" Video file associated with this item.
208-299	Reserved for future OPC use.

Recommended Properties Related to Alarm and Condition Values

PropID	Description
300	"Condition Status" The current alarm condition status associated with the item.
301	"Alarm Quick Help" A short text providing a brief set of instructions for the operator to follow when this alarm occurs.
302	"Alarm Area List" An array of texts indicating the plant or alarm areas which include this item.
303	"Primary Alarm Area" A text indicating the primary plant or alarm area including this item.
304	"Condition Logic" An arbitrary test describing the test being performed.
305	"Limit Exceeded" For multistate alarms, the condition exceeded.
306	"Deadband"
307	"HiHi Limit"
308	"Hi Limit"
309	"Lo Limit"
310	"LoLo Limit"
311	"Rate of Change Limit"
312	"Deviation Limit"
313-4999	Reserved for future OPC use.

See Also

More About

- "OPC DA Server Item Properties" on page B-2
- "OPC Item Property Set" on page B-3
- "OPC Specific Properties" on page B-4

OPC HDA Item Attributes

OPC HDA Item Attributes

- **Data Type** — Specifies the data type for an item. See the definition of a particular Variant for valid values.

Comparison of MATLAB and COM Variant Data Types

MATLAB Data Type	OPC Server Data Type (COM Variant Type)
double	VT_R8
single	VT_R4
char	VT_BSTR
logical	VT_BOOL
uint8	VT_UI1
uint16	VT_UI2
uint32	VT_UI4
uint64	VT_UI8
int8	VT_I1
int16	VT_I2
int32	VT_I4
int64	VT_I8
cell	N/A
struct	N/A
object	N/A
N/A	VT_DISPATCH
N/A	VT_BYREF
double	VT_EMPTY

- **Description** — Describes the item.
- **Eng Units** — Specifies the label to use in displays to define the units for the item (e.g., kg/sec).
- **Stepped** — Specifies whether data from the history repository should be displayed as interpolated (sloped lines between points) or stepped (vertically-connected horizontal lines between points) data. Value of 0 indicates interpolated.
- **Archiving** — Indicates whether historian is recording data for this item (0 means no).
- **Derive Equation** — Specifies the equation to be used by a derived item to calculate its value. This is free-form text.
- **Node Name** — Specifies the machine which is the source for the item. This is intended to be the broadest category for defining sources. For an OPC Data Access Server source, this is the node name or IP address of the server. For non-OPC sources, the meaning of this field is server-specific.
- **Process Name** — Specifies the process which is the source for the item. This is intended to the second-broadest category for defining sources. For an OPC DA server, this would be the registered server name. For non-OPC sources, the meaning of this field is server-specific.
- **Source Name** — Specifies the name of the item on the source. For an OPC DA server, this is the ItemID. For non-OPC sources, the meaning of this field is server-specific.

- **Source Type** — Specifies what sort of source produces the data for the item. For an OPC DA server, this would be "OPC". For non-OPC sources, the meaning of this field is server-specific.
- **Normal Maximum** — Specifies the upper limit for the normal value range for the item. It is used for trend display default scaling and exception deviation limit calculations.
- **Normal Minimum** — Specifies the lower limit for the normal value range for the item. It is used for trend display default scaling and exception deviation limit calculations.
- **ItemID** — Specifies the item ID.
- **Max Time Interval** — Specifies the maximum interval between data points in the history repository regardless of their value change. A new value shall be stored in history whenever the specified number of seconds have passed since the last value stored for the item.
- **Min Time Interval** — Specifies the minimum interval between data points in the history repository regardless of their value change. A new value shall not be stored in history unless the specified number of seconds have passed since the last value stored for the item.
- **Exception Deviation** — Specifies the minimum amount that the data for the item must change in order for the change to be reported to the history database.
- **Exception Dev Type** — Specifies whether the exception deviation is given as an absolute value, percent of span, or percent of value. The span is defined as High Entry Limit – Low Entry Limit.
- **High Entry Limit** — Specifies the highest valid value for the item. A value for the item that is above this limit cannot be entered into history. This is the top of the span.
- **Low Entry Limit** — Specifies the lowest valid value for the item. A value for the item that is below this limit cannot be entered into history. This is the zero for the span. What follows is a list describing the OPC specified attributes which may be supported by the server.

See Also

Functions
`readItemAttributes`

More About

- “Read Item Attributes” on page 13-9

Functions

addgroup

Add data access group to `opcda` object

Syntax

```
GrpObj = addgroup(DAObj)
GrpObj = addgroup(DAObj,GName)
GrpObj = addgroup(DAObj,GName,GrpType)
```

Description

`GrpObj = addgroup(DAObj)` adds a group to the `opcda` object `DAObj`. A group is a container for a client to organize and manipulate data items. Typically, you create different groups to support different update rates, activation status, callbacks, etc.

If `DAObj` is already connected to the server when `addgroup` is called, a group name is requested from the server. If the server does not supply a group name, or the object is not connected to a server, a unique name is automatically assigned to `GrpObj`. The unique name follows the convention `'groupN'` where `N` is an integer. You can change this name by modifying the group's `Name` property.

`GrpObj = addgroup(DAObj,GName)` adds a group to the OPC data access object `DAObj` with the group name given by `GName`. The group name must be unique among other group names within `DAObj`.

`GrpObj = addgroup(DAObj,GName,GrpType)` adds a group to the `opcda` object `DAObj` with the group type specified by `GrpType`, either `'private'` or `'public'`.

You can add items to `GrpObj` using the `additem` function, if the group type is `'private'`. For a public group, the items are already defined, and are automatically created when you connect to the public group using `addgroup`.

Examples

Create an OPC DA Client and Add Groups

Create an OPC DA client and add groups to it.

Create an `opcda` client.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
```

Create a group using a default group name.

```
grp1 = addgroup(da);
```

Add another group, providing its name.

```
grp2 = addgroup(da,'AddgroupEx');
```

Input Arguments

`DAObj` — OPC DA client

OPC DA client object

OPC DA client , specified as an OPC DA client object. You create the client object with the `opcda` function.

Example: `DAObj = opcda()`

`GName` — Group name

char | string

Group name, specified as a character vector or string. The group name must be unique within the OPC DA client object.

Example: `'group1'`

Data Types: `char` | `string`

`GrpType` — Group type

`'private'` (default) | `'public'`

Group type, specified as `'private'` or `'public'`. If `GrpType` is `'private'` (the default), the group is configured to be private to `DAObj`, and no other client connected to the OPC server can access that group. If `GrpType` is `'public'`, a connection is made to the server's public group named `GName`. To make a connection to a public group named `GName`, that group must exist on the server as a public group. You create public groups on the server using the `makepublic` function. Note that some servers do not support public groups; you can verify whether a server supports public groups by using `opcserverinfo(DAObj)` and checking the `SupportedInterfaces` field for the `IOPCServerPublicGroups` interface.

Example: `'public'`

Data Types: `char` | `string`

Output Arguments

`GrpObj` — Data access group

`dagroup` object

Data access group, returned as a `dagroup` object, with properties described in dagroup Object Properies Properties.

By default, `GrpObj` has its Active property set to `'on'`, GroupType set to `'private'`, and the Subscription property set to `'on'`.

Version History

Introduced before R2006a

See Also

Functions
`additem` | `opcda` | `opcserverinfo`

Properties
dagroup Object Properies Properties

additem

Add data access items to `dagroup` object

Syntax

```
IObj = additem(GObj,'IName')
IObj = additem(GObj,'IName','DataType')
IObj = additem (GObj,'IName','DataType','Active')
```

Description

`IObj = additem(GObj,'IName')` adds items to the group object `GObj` with fully qualified item IDs given by `IName`. The object `IObj` is the created item object or objects, with properties described in daitem Object Properties Properties. You specify `IName` as a single item ID or as a cell array of item IDs.

The `daitem` object provides a connection to a data variable in the physical device and returns information about the data variable, such as its value, quality, and time stamp. Note that you cannot add a given item to the same group more than once. However, you can add the same item to different groups.

By default, `IObj` is active; that is, if the group’s Subscription property is `on`, the item's Value, Quality, and TimeStamp properties will be updated at the group's UpdateRate.

Servers often require item IDs to be specified in the correct case. You can use the `serveritems` function to find valid item IDs.

Note You cannot add items to a public group. A public group has a fixed set of item IDs common to all clients sharing that group. The `GroupType` property of a `dagroup` object indicates the type of group.

`IObj = additem(GObj,'IName','DataType')` adds items to the group object `GObj` with the requested data type given by *`'DataType'`*. You specify *`'DataType'`* as a cell array of character vectors, one for each item ID. *`'DataType'`* is the data type in which the item's value is stored in the MATLAB workspace. The supported data types are `'logical'`, `'int8'`, `'uint8'`, `'int16'`, `'uint16'`, `'int32'`, `'uint32'`, `'single'`, `'double'`, `'char'`, and `'date'`. Note that if the requested data type is rejected by the server, the item is not added. The requested data type is stored in the `DataType` property. The canonical data type (the data type used by the server to store the item value) is stored in the `CanonicalDataType` property.

`IObj = additem (GObj,'IName','DataType','Active')` adds items to the group object `GObj` with active status given by *`'Active'`*. You specify *`'Active'`* as a cell array of character vectors, one for each item ID. *`'Active'`* can be `'on'` or `'off'`. The active status is stored in the `Active` property.

Examples

Add Items to a Group

Add items with different attributes to a group.

Create a client and a group.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExAddItem');
```

Add two items with their canonical data types.

```
itm = additem(grp, {'Random.Real4','Random.Real8'});
```

Add an item with a `'double'` data type.

```
itmDbl = additem(grp,'Random.Int2','double');
```

Add an inactive item.

```
itmInact = additem(grp,'Random.UInt4','double','off');
```

Version History

Introduced before R2006a

See Also

Functions
`getnamespace` | `serveritems`

Properties
daitem Object Properties Properties

arrayHasSameTimeStamp

Class: `opc.hda.Data`
Package: `opc.hda`

True if all elements of OPC HDA data object have same time stamp vector

Syntax

`tf = arrayHasSameTimeStamp(dObj)`

Description

`tf = arrayHasSameTimeStamp(dObj)` returns true if all the elements of `dObj` have the same time stamp.

Use `tsunion` to ensure that the time stamps of an OPC HDA data object are the same.

Examples

Load the OPC HDA example data file and see if the `hdaDataSmall` object has the same time stamps in all elements:

```
load opcSampleHdaData;
tf = arrayHasSameTimeStamp(hdaDataSmall);
```

Form a new data set using `tsunion`, and check the time stamps again:

```
hdaUnion = tsunion(hdaDataSmall);
tfU = arrayHasSameTimeStamp(hdaUnion)
```

See Also
`tsunion`

browsenamespace

Graphically browse OPC DA server name space

Syntax

```
ItmList = browsenamespace(DaObj)
ItmList = browsenamespace(DaObj,ItmListInit)
ItmList = browsenamespace(DaObj,ItmListInit,true)
```

Description

`ItmList = browsenamespace(DaObj)` opens a graphical name space browser for the OPC Data Access Client object `DaObj`. The graphical interface lets you construct a list of items and return a list of those fully qualified item IDs to `ItmList`. You can use `ItmList` to add items to a Group object using `additem`. The name space is retrieved from the server incrementally, as needed.

`ItmList = browsenamespace(DaObj,ItmListInit)` lets you specify an initial list of item IDs to augment.

`ItmList = browsenamespace(DaObj,ItmListInit,true)` loads the entire name space into the dialog box.

Examples

Browse Local Matrikon Server for OPC DA Items

Connect to the local Matrikon Simulation server and browse for items.

```
DaObj = opcda('localhost','Matrikon.OPC.Simulation');
connect(DaObj);
ItmList = browsenamespace(DaObj);
```

Input Arguments

`DaObj` — OPC DA client
OPC DA client object

OPC DA client, specified as an OPC DA client object.

`ItmListInit` — Initial list of OPC DA items
character vector, string, or cell array

Initial list of OPC DA items, specified as a character vector, string, or cell array that identifies the item IDs. When the browser opens, these items are already included in the selected list.

Data Types: `char` | `string` | `cell`

`true` — Indicator to load entire name space
true

Indicator to load the entire name space, specified as `true`. Use this option only if your server does not support partial name space browsing.

Data Types: `logical`

Output Arguments

`ItmList` — List of OPC DA item IDs
char vector or cell array of char vectors

List of OPC DA item IDs, returned as a character vector or cell array of character vectors. Each character vector indicates a selected OPC DA item ID.

Version History

Introduced in R2013a

See Also

`getnamespace` | `addgroup` | `additem`

browseNameSpace

Package: `opc.hda`

Graphically browse OPC HDA server name space

Syntax

```
ItmList = browseNameSpace(HdaObj)
ItmList = browseNameSpace(HdaObj,ItmListInit)
ItmList = browseNameSpace(HdaObj,ItmListInit,true)
```

Description

`ItmList = browseNameSpace(HdaObj)` opens a graphical name space browser for the OPC HDA client object `HdaObj`. Use the graphical interface to construct a list of items and return a list of those fully qualified item IDs in `ItmList`. Use `ItmList` to retrieve data for those items with function `readRaw`, `readProcessed`, `readAtTime`, or `readModified`.

The name space is retrieved from the server incrementally, as needed.

`ItmList = browseNameSpace(HdaObj,ItmListInit)` lets you specify an initial list of item IDs to be augmented.

`ItmList = browseNameSpace(HdaObj,ItmListInit,true)` loads the entire name space into the dialog.

Examples

Browse Local Matrikon Server for OPC HDA Items

Connect to the local Matrikon Simulation server and browse for items.

```
HdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(HdaObj);
ItmList = browseNameSpace(HdaObj);
```

Input Arguments

`HdaObj` — OPC HDA client
OPC HDA client object

OPC HDA client, specified as an OPC HDA client object.

`ItmListInit` — Initial list of OPC HDA items
character vector, string, or cell array of character vectors

Initial list of OPC HDA items, specified as a character vector, string, or cell array that identifies the item IDs. When the browser opens, these items are already included in the selected list.

Data Types: `char` | `string` | `cell`

`true` — Indicator to load entire name space
true

Indicator to load the entire name space, specified as `true`. Use this option only if your server does not support partial name space browsing.

Data Types: `logical`

Output Arguments

`ItmList` — List of OPC HDA item IDs
char vector or cell array of char vectors

List of OPC HDA item IDs, returned as a character vector or cell array of character vectors. Each character vector indicates a selected OPC HDA item ID.

Version History

Introduced in R2013a

See Also

Functions
`getNameSpace (opchda)` | `readRaw` | `readProcessed` | `readAtTime` | `readModified`

browseNamespace

Package: `opc.ua`

Graphically browse name space and select nodes from OPC UA server

Syntax

```
NodeList = browseNamespace(UaClient)
NodeList = browseNamespace(UaClient,Nodes)
```

Description

`NodeList = browseNamespace(UaClient)` opens the Browse Name Space dialog box for OPC UA client object `UaClient`. Using this browser, you can construct a list of nodes, and return an array of those nodes in `NodeList`. You can use `NodeList` to retrieve data for those items using `read`, `readHistory`, `readProcessed`, `readAtTime`, or `readModified`.

The name space is retrieved from the server incrementally. `UaClient` must be connected when you call this function.

`NodeList = browseNamespace(UaClient,Nodes)` allows you to specify an initial list of `Nodes` to be supplemented. If you cancel the browsing by pressing the **Cancel** button, then `NodeList` will be empty.

Examples

Create Initial List of Nodes

This example shows how to create a list of nodes from the OPC UA name space. After selecting the nodes you want in the dialog box, click **OK**.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
NodeList1 = browseNamespace(UaClient)
```

Supplement List of Nodes

This example shows how to add to a list of nodes from the OPC UA name space. The Browse Name Space dialog box opens with the nodes of `NodeList1` already selected.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
NodeList1 = browseNamespace(UaClient)
```

```
% Some time later
NodeList2 = browseNamespace(UaClient,NodeList1)
```

Input Arguments

`UaClient` — OPC UA client
OPC UA client object

OPC UA client specified as an OPC UA client object

`Nodes` — List of nodes
array of node objects

List of nodes returned as an array of node objects. For information on node object functions and properties, type

```
help opc.ua.Node
```

Output Arguments

`NodeList` — List of nodes
array of node objects

List of nodes returned as an array of node objects. For information on node object functions and properties, type

```
help opc.ua.Node
```

Version History

Introduced in R2015b

See Also

Functions
`getNamespace` | `readValue` | `readHistory` | `readProcessed` | `readAtTime` | `writeValue`

cancelasync

Cancel asynchronous read and write operations

Syntax

```
cancelasync(GObj)
cancelasync(GObj,TransID)
```

Description

`cancelasync(GObj)` cancels all asynchronous read and write operations that are in progress for the group object specified by `GObj`. This function is asynchronous and does not block the MATLAB command line.

After `cancelasync` cancels the in-progress asynchronous operations, the OPC server generates a cancel async event. If you specify a callback function file for the `CancelAsyncFcn` property, the callback function executes when this event occurs.

`cancelasync(GObj,TransID)` cancels the asynchronous operations, specified by the transaction IDs given by `TransID`. You can cancel specific asynchronous requests using this syntax.

Examples

Cancel Asynchronous Read Operation

Start an asynchronous read and then cancel it.

Create a connected client, group, and items.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'CancelAsyncEx');
additem(grp, {'Random.Real8','Random.Real4'});
```

Request an asynchronous read operation and then immediately cancel that request.

```
tid = readasync(grp); cancelasync(grp,tid);
```

Version History

Introduced before R2006a

See Also

Functions
`readasync` | `writeasync`

cleareventlog

Clear event log, discarding all events

Syntax

```
cleareventlog(DAObj)
```

Description

`cleareventlog(DAObj)` clears the event log for `opcda` object `DAObj`. `DAObj` can be an array of objects. `cleareventlog` also discards any events stored in the `EventLog` property of the objects.

Examples

Create a connected client and configure a group with two items:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ClearEventLogEx');
itm1 = additem(grp,'Random.Real8');
itm2 = additem(grp,'Triangle Waves.UInt1');
```

Run a 10-second logging task, and after 5 seconds perform an asynchronous read of the group:

```
grp.UpdateRate = 1;
grp.RecordsToAcquire = 10;
start(grp);
pause(5);
tid = readasync(grp);
wait(grp);
```

Examine the event log size:

```
el = da.EventLog
```

Clear the event log:

```
cleareventlog(da)
el2 = da.EventLog
```

Version History

Introduced before R2006a

clonegroup

Clone group into new private group on same client

Syntax

```
NewGObj = clonegroup(GObj,'NewName')
```

Description

`NewGObj = clonegroup(GObj,'NewName')` clones the `dagroup` object specified by `GObj`, making a private group `NewGObj` with name `NewName`. `NewName` must be a unique group name. `GObj` can be a private group or a public group.

The new group `NewGObj` is independent of the original group, but with the same parent (`opcda` object) and the same items as that group. All the group and item properties are duplicated with the exception of the following:

- The `Active` property is configured to `'off'`.
- The `GroupType` property is configured to `'private'`.

Not all OPC data access servers support the cloning of groups. To use this functionality, your server must support public groups. If you try to clone a group on a server that does not support public groups, an error is generated. To verify that a server supports public groups, use the `opcserverinfo` function on the client connected to that server: Look for an entry `'IOPCPublicGroups'` in the `'SupportedInterfaces'` field.

You use `clonegroup` primarily when you want to create a private duplicate of a public group that you can then modify. If you want to create a copy of a group in another client, use the `copyobj` function.

Examples

Create a fictitious client, and configure a group with two items. Do not connect to the server.

```
da = opcda('localhost','Dummy.Server');
grp1 = addgroup(da,'OriginalGroup');
itm1 = additem(grp1,'Device1.Item1');
itm2 = additem(grp1,'Device1.Item2');
```

Clone the group.

```
grp2 = clonegroup(grp1,'ClonedGroup');
```

Version History

Introduced before R2006a

See Also

Functions
`copyobj` | `makepublic`

connect

Package:

Connect client object to OPC server

Syntax

```
connect(Obj)
```

Description

`connect(Obj)` connects the `opcda` or `opchda` object `Obj` to the OPC server that specified by the object `Host` and `ServerID` properties. When you connect `Obj`, its `Status` property takes the value `'connected'`. You can disconnect `Obj` from the server with the `disconnect` function, which sets the `Status` property value to `'disconnected'`.

If `Obj` is an array of objects and the function cannot connect some of these objects, it generates a warning. If the function cannot connect any of the objects, it generates an error.

It is possible to create `opcda` groups and items before connecting to the server. However, servers impose restrictions on client group and item names. Therefore, if you create a group hierarchy and then connect to the server, `connect` automatically deletes groups or items that the server cannot support, and issues a warning message.

Examples

Connect OPC DA Client to Sever

Create a Data Access client and connect to the server.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
```

Connect OPC HDA Client to Server

Create an HDA client for the Matrikon Simulation Server and connect to the server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
```

Input Arguments

`Obj` — OPC client object

opcda object | opchda object

OPC client object, specified as an opcda object, opchda object, or an array of objects.

Example: `opchda()`

Version History

Introduced before R2006a

See Also

Functions

`disconnect` | `isConnected`

connect

Package: `opc.ua`

Connect OPC UA client to server

Syntax

```
connect(UaClient)
connect(UaClient, UserName, Password)
connect(UaClient,PublicKeyFilename,PrivateKeyFileName,PrivateKeyPassword)
```

Description

`connect(UaClient)` connects the OPC UA client `UaClient` to its referenced server using anonymous user authentication.

`connect(UaClient, UserName, Password)` connects the OPC UA Client `UaClient` to its server using username and password authentication. The `UserName` and `Password` arguments must be provided, although the `Password` field can be empty.

`connect(UaClient,PublicKeyFilename,PrivateKeyFileName,PrivateKeyPassword)` connects the OPC UA Client `UaClient` to its server using the User Certificate stored in the public and private key files referenced by `PublicKeyFilename` and `PrivateKeyFilename`. `PrivateKeyPassword` is the password used to protect the Private Key File. Private Key Files for OPC must be password protected. The files must be in `.DER` format.

When the client successfully connects to the server, the client object `Status` property is set to `'Connected'`, the first level of the server namespace is retrieved, and various essential properties of the client are read from the server.

If `UaClient` is a vector of clients, and some but not all clients can connect, a warning is issued. If no clients can connect, an error is generated. You can only connect a vector of clients using the same username and password, or the same certificate parameters. If you need to use different usernames and passwords for different servers, call `connect` on each of the clients individually.

Examples

Connect OPC UA Client to Server

Locate an OPC UA server and connect a client to it.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s(1));
connect(UaClient);
```

Check the connection status.

```
isConnected(UaClient)
```

Input Arguments

`UaClient` — OPC UA client
OPC UA client object

OPC UA client, specified as an OPC UA client object or array of objects.

Example: `opcua()`

Version History

Introduced in R2015b

See Also

Functions
`disconnnect` | `isConnected` | `opcua`

copyobj

Make copy of OPC data access object

Syntax

```
NewObj = copyobj(Obj)
NewObj = copyobj(Obj, ParentObj)
```

Description

`NewObj = copyobj(Obj)` makes a copy of all the objects in `Obj`, and returns them in `NewObj`. `Obj` can be a scalar OPC object, or a vector of toolbox objects.

`NewObj = copyobj(Obj, ParentObj)` makes a copy of the objects in `Obj` inside the parent object `ParentObj`. `ParentObj` must be a valid scalar parent object for `Obj`. If any objects in `Obj` cannot be created in `ParentObj`, a warning will be generated.

A copied toolbox object contains new versions of all children, their children, and any parents that are required to construct that object. A copied object is different from its parent object in the following ways:

- The values of read-only properties will not be copied to the new object. For example, if an object is saved with a Status property value of `'connected'`, the object will be recreated with a `Status` property value of `'disconnected'` (the default value). You can use `propinfo` to determine if a property is read-only. Specifically, a connected `opcda` object is copied in the disconnected state, and a copy of a logging `dagroup` object is not reset to the logging state.
- A copied `dagroup` object that has records in memory from a logging session is copied without those records.

  OPC HDA objects do not support `copyobj`.

Examples

Create a connected Data Access client with a group containing an item:

```
da1 = opcda('localhost', 'Matrikon.OPC.Simulation');
connect(da1);
grp1 = addgroup(da1, 'CopyobjEx');
itm1 = additem(grp1, 'Random.Real8');
```

Copy the client object. This also copies the group and item objects.

```
da2 = copyobj(da1);
grp2 = da2.Group
```

Change the first group name, and note that the second group name is unchanged:

```
grp1.Name = 'NewGroupName';
grp2.Name
```

Version History

Introduced before R2006a

See Also

`obj2mfile` | `propinfo`

delete

Package:

Remove OPC objects from memory

Syntax

```
delete(Obj)
```

Description

`delete(Obj)` removes the OPC object `Obj` from memory. `Obj` can be an array of objects. A deleted object becomes invalid and you cannot reconnect it to the server after it has been deleted, so you should remove references to that object from the workspace with the `clear` command. Deleting an object that contains children (groups or items) also deletes these children, so you should remove references to these children.

If multiple references to a toolbox object exist in the workspace, then deleting one object invalidates the remaining references.

If `Obj` is an `opcda` object connected to the server, `delete` disconnects and deletes the object.

Examples

Create an OPC HDA Client, delete the object, and clear the variable from the workspace:

```
hdaObj = opchda('localhost', 'Matrikon.OPC.Simulation');
delete(hdaObj);
clear hdaObj
```

Delete a group and its children from memory:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'DeleteEx');
itm = additem(grp,'Random.Real4');
r = read(grp)
delete(grp);    % deletes itm as well
clear grp itm
```

Version History

Introduced before R2006a

See Also

`clear` | `disconnect` | `isvalid` | `opc.hda.reset`

disconnect

Package:

Disconnect client object from OPC server

Syntax

```
disconnect(Obj)
```

Description

`disconnect(Obj)` disconnects the OPC client object `Obj` from the server. `Obj` can be an array of objects.

If the disconnection from the server was successful, the function sets the `Obj` property Status value to `'disconnected'`. You can reconnect `Obj` to the server with the `connect` function.

If `Obj` is an array of objects and the function cannot disconnect some of the objects from the server, it disconnects the remaining objects in the array and issues a warning. If the function can disconnect none of the objects from their server, it generates an error.

Examples

Disconnect from OPC DA Server

Create an OPC data access client and connect to the server:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
da.Status
```

Disconnect from the server:

```
disconnect(da);
da.Status
```

Disconnect from OPC HDA Server

Create an OPC HDA client for the Matrikon Simulation Server and connect to the server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
```

Check the status of the connection.

```
hdaObj.Status
```

Disconnect from the server and check the status again.

```
disconnect(hdaObj);
hdaObj.Status
```

Version History

Introduced before R2006a

See Also

Functions
`connect` | `isConnected` | `propinfo`

disconnect

Package: `opc.ua`

Disconnect OPC UA client from server

Syntax

`disconnect(UaClient)`

Description

`disconnect(UaClient)` disconnects the OPC UA client `UaClient` from its server, and sets the client `Status` property to `'Disconnected'`.

Examples

Disconnect an OPC UA client and view its connection status.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
UaClient.Status
```

```
Connected
```

```
disconnect(UaClient);
UaClient.Status
```

```
Disconnected
```

Version History

Introduced in R2015b

See Also

`connect` | `isConnected` | `opcua`

disp

Summary of information for OPC objects

Syntax

```
Obj
disp(Obj)
```

Description

`Obj` or `disp(Obj)` displays summary information for the OPC object `Obj`.

If `Obj` is an array of objects, `disp` outputs a table of summary information about the objects in the array.

Summary information includes the following information as appropriate for each item in `dObj`.

- `ItemID`: The item ID for that element.
- `Value`: The number and data type of the values for that element.
- `Start TimeStamp`: The time of the first value in the element. The time is displayed in the format specified by the OPC date display format that can you set using `opc.setDateDisplayFormat`
- `End TimeStamp`: The time of the last value in the element.
- `Quality`: The number of unique qualities contained in the element. If all values have the same quality, that HDA quality is displayed.

You can get more information about a OPC HDA data objects by using the `showValues` method.

Alternatively, you can display summary information for `Obj` by excluding the semicolon when:

- Creating a toolbox object, using the `opcda`, `addgroup`, or `additem` functions
- Configuring property values using dot notation

Examples

Display the summary of a data access client:

```
da = opcda('localhost', 'My.Server.1')
```

```
da =

Summary of OPC Data Access Client Object: localhost/My.Server.1

   Server Parameters
      Host      : localhost
      ServerID  : My.Server.1
      Status    : disconnected
      Timeout   : 10 seconds

   Object Parameters
```

```
      Group     : 0-by-1 dagroup object
      Event Log : 0 of 1000 events
```

Display the summary information for an array of data access clients:

```
da2 = opcda('localhost', 'My.Second.Server.1');
[da da2]
```

```
   OPC Data Access Object Array:

   Index:  Status:         Name:
   1       disconnected    localhost/My.Server.1
   2       disconnected    localhost/My.Second.Server.1
```

Load the OPC HDA example data file and display the `hdaDataSmall` object:

```
load opcSampleHdaData;
disp(hdaDataSmall)
```

Version History

Introduced before R2006a

See Also

`addgroup` | `additem` | `opcda` | `showValues`

double

Package: `opc.hda`

Convert OPC HDA data object array to double matrix

Syntax

```
Vdouble = double(DObj)
```

Description

`Vdouble = double(DObj)` converts the OPC HDA data object array `DObj` into a matrix of data type `double`.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to Matrix of doubles

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create a matrix of type `double` from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vdouble = double(dUnion);
```

Input Arguments

`DObj` — OPC HDA data
OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vdouble` — OPC HDA data values
matrix of `double` type

OPC HDA data values, returned as a matrix of `double` type. `Vdouble` is constructed as an `M`-by-`N` matrix of `double` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

exportClientCertificate

Package: `opc`

Copy OPC UA client application certificates to file

Syntax

```
fileName = opc.ua.exportClientCertificate
fileName = opc.ua.exportClientCertificate("SHA1")
fileName = opc.ua.exportClientCertificate("SHA256")
opc.ua.exportClientCertificate("SHA1",FileName)
opc.ua.exportClientCertificate("SHA256",FileName)
```

Description

`fileName = opc.ua.exportClientCertificate` copies the toolbox SHA256 UA Client Application Certificate to the file `MATLAB_OPCToolbox_SHA256.der` in the user folder. The full path to the file is returned in `fileName`.

`fileName = opc.ua.exportClientCertificate("SHA1")` copies the toolbox SHA1 UA Client Application Certificate to the file `MATLAB_OPCToolbox_SHA1.der` in the user folder. Note that SHA1 is considered insecure by the OPC Foundation, and this certificate should be used only for backward compatibility. The full path to the file is returned in `fileName`.

`fileName = opc.ua.exportClientCertificate("SHA256")` copies the toolbox SHA256 UA Client Application Certificate to the file `MATLAB_OPCToolbox_SHA256.der` in the user folder. The full path to the file is returned in `fileName`.

`opc.ua.exportClientCertificate("SHA1",FileName)` or `opc.ua.exportClientCertificate("SHA256",FileName)` copies the corresponding toolbox UA Client Application Certificate to the file given by `FileName`. If the full path to `FileName` does not exist, the function attempts to create it. You can use the generated file to register the Client Application Certificate with any servers that require trusted certificates. The Client Application Certificate is exported in `.der` format.

Examples

Export Client Certificate

Export the SHA256 UA Client Application Certificate.

```
fName = opc.ua.exportClientCertificate("SHA256");
```

The generated file is named `MATLAB_OPCToolbox_SHA256.der` in the folder identified in `fName`.

Input Arguments

`FileName` — Path to generated certificate file
string | char

Full path to generated certificate file, specified as a string or character vector.

Example: `"C:\st4\certfile.der"`

Data Types: `char` | `string`

Output Arguments

`fileName` — File name of exported certificate
char

File name with full path to location of exported certificate file.

Version History
Introduced in R2020a

See Also

Functions
`opcua` | `setSecurityModel`

Topics
“OPC UA Security” on page 17-7
“OPC UA Certificate Management” on page 17-9

findDescription

Package: `opc.hda`

Locate OPC HDA servers with particular description

Syntax

```
ind = findDescription(SIObj,'DescStr')
```

Description

`ind = findDescription(SIObj,'`*`DescStr`*`')` returns the indices of the OPC HDA ServerInfo elements in `SIObj`, where the `Description` property starts with `'`*`DescStr`*`'`.

Examples

Locate all servers on the local host, with the description starting `'Matrikon'`.

```
siObj = opchdaserverinfo('localhost');
ind = findDescription(siObj,'Matrikon');
siMatrikon = siObj(ind)
```

See Also

Functions
`opchdaserverinfo`

findDescription

Package: `opc.ua`

Find OPC UA servers containing specified description

Syntax

```
ServerList = findDescription(Servers,DescStr)
```

Description

`ServerList = findDescription(Servers,DescStr)` searches among `Servers` and returns only those OPC UA servers whose `Description` property contains the character vector or string `DescStr`.

Examples

Find all sample servers from the local host.

```
localServers = opcuaserverinfo('localhost');
sampleServers = findDescription(localServers,'Sample')
```

```
sampleServers = 
OPC UA ServerInfo 'UA Sample Server':

   Connection Information
    Hostname: 'HOST2241'
        Port: 51210
```

Version History

Introduced in R2015b

See Also

Functions
`opcuaserverinfo`

findNodeById

Package: opc.ua

Find OPC UA server nodes by namespace index and identifier

Syntax

```
FoundNode = findNodeById(NodeList,NsInd,Id)
```

Description

`FoundNode = findNodeById(NodeList,NsInd,Id)` searches the nodes in `NodeList` for a node whose `NamespaceIndex` and `Identifier` properties match `NsInd` and `Id`, respectively. `NsInd` must be an integer, and `Id` must be a character vector, string, or integer.

This function might query the server for further descendants (children) of `NodeList`.

Examples

Find the `ServerCapabilities` node (`Index` 0, `Identifier` 2268) of the OPC UA server on the local host.

```
UaClient = opcua('localhost',51210);
connect(UaClient);
capabilitiesNode = findNodeById(UaClient.Namespace,0,2268)
```

```
capabilitiesNode = 

OPC UA Node:

   Node Information:
                      Name: 'ServerCapabilities'
               Description: 'Describes capabilities supported by the server.'
            NamespaceIndex: 0
                Identifier: 2268
                  NodeType: 'Object'

   Hierarchy Information:
                    Parent: Server
                  Children: 14
```

Version History

Introduced in R2015b

See Also

Functions
`findNodeByName` | `opcua`

findNodeByName

Package: `opc.ua`

Find OPC UA server nodes by name

Syntax

```
FoundNodes = findNodeByName(NodeList,NodeName)
FoundNodes = findNodeByName(NodeList,NodeName,'-once')
FoundNodes = findNodeByName(NodeList,NodeName,'-partial')
FoundNodes = findNodeByName(NodeList,NodeName,'-once','-partial')
```

Description

`FoundNodes = findNodeByName(NodeList,NodeName)` searches the descendants of `NodeList` for all nodes whose `Name` property matches `NodeName`. The search among all nodes, including `NodeList`, is not case sensitive.

`FoundNodes = findNodeByName(NodeList,NodeName,'-once')` stops searching when one node has been found.

`FoundNodes = findNodeByName(NodeList,NodeName,'-partial')` finds all nodes that start with `NodeName`.

`FoundNodes = findNodeByName(NodeList,NodeName,'-once','-partial')` finds only the first partial match.

This function might query the server for further descendants (children) of `NodeList`.

Examples

Find the `ServerCapabilities` node from the server node.

```
UaClient = opcua('localhost',51210);
connect(UaClient);
serverNode = findNodeByName(UaClient.Namespace,'Server','-once');
capabilitiesNode = findNodeByName(serverNode,'ServerCapabilities')
```

```
capabilitiesNode =
OPC UA Node:

   Node Information:
                      Name: 'ServerCapabilities'
               Description: 'Describes capabilities supported by the server.'
            NamespaceIndex: 0
                Identifier: 2268
                  NodeType: 'Object'

   Hierarchy Information:
                    Parent: Server
                  Children: 14
```

Version History

Introduced in R2015b

See Also

Functions
`findNodeById` | `opcua`

flatnamespace

Flatten hierarchical OPC name space

Syntax

```
FNS = flatnamespace(NS)
```

Description

`FNS = flatnamespace(NS)` flattens the hierarchical name space `NS`, by recursively removing all information in the `Nodes` fields of `NS` and placing that information into additional entries in the root structure of `FNS`. You obtain a hierarchical name space using the `'hierarchical'` flag in `getnamespace`.

Examples

Retrieve the name space for the Matrikon Simulation Server, and then flatten the name space:

```
da = opcda('localhost', 'Matrikon.OPC.Simulation');
connect(da);
hierNS = getnamespace(da)
flatNS = flatnamespace(hierNS)
```

Version History

Introduced before R2006a

See Also

`getnamespace` | `serveritems`

flush

Package: `icomm.mqtt`

Clear received MQTT messages

Syntax

```
flush(mqttClient)
flush(mqttClient,Topic=mqttTopic)
```

Description

`flush(mqttClient)` clears all received messages from all subscribed topics in the specified MQTT client.

`flush(mqttClient,Topic=mqttTopic)` clears all received messages from the specified MQTT topic.

Note that the `read` function also clears messages after reading them into MATLAB, but the `peek` function does not.

Examples

Flush Messages from MQTT Topic

View a recent message and then flush all messages.

```
peek(mqttClient,Topic="TopMW01")
```

```
ans =

  1×2 timetable

            Time              Topic           Data
    ____________________    _________    _______________

    14-Dec-2021 16:29:09    "TopMW01"    "Hello World 3"
```

```
flush(mqttClient)
peek(mqttClient)
```

```
Warning: No data available to peek for topic "TopMW01".

ans =

  0×2 empty timetable
```

Input Arguments

`mqttClient` — MQTT client
`Client` object

MQTT client specified as an `icomm.mqtt.Client` object, created with the `mqttclient` function.

Example: `mqttClient = mqttclient()`

Data Types: `object`

`mqttTopic` — MQTT topic
string | char

MQTT topic to flush messages from, specified as a string or character vector.

Example: `"trubits/mqTop48"`

Data Types: `string` | `char`

Version History
Introduced in R2022a

See Also

Functions
`mqttclient` | `subscribe` | `unsubscribe` | `peek`

flushdata

Remove all logged data records associated with `dagroup` object

Syntax

```
flushdata(GObj)
```

Description

`flushdata(GObj)` removes all records associated with the `dagroup` object `GObj` from the toolbox engine, and sets `RecordsAvailable` to `0` for that object.

`GObj` can be a scalar `dagroup` object, or a vector of `dagroup` objects.

Examples

Create a connected client and configure a group with two items:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ClearEventLogEx');
itm1 = additem(grp,'Random.Real8');
```

Acquire 10 records using a logging task:

```
grp.UpdateRate = 0.5;
grp.RecordsToAcquire = 10;
start(grp);
wait(grp);
```

Examine the records available:

```
recordCount1 = grp.RecordsAvailable
```

Flush all data from the client:

```
flushdata(grp)
recordCount2 = grp.RecordsAvailable
```

Version History

Introduced before R2006a

See Also

`getdata` | `peekdata` | `start` | `stop`

genslread

Generate Simulink OPC Read block from MATLAB group object

Syntax

```
BlkPath = genslread(GrpObj)
BlkPath = genslread(GrpObj,DestSys)
```

Description

`BlkPath = genslread(GrpObj)` generates an OPC Read block from the `dagroup` object `GrpObj`, and places the block in a new Simulink model. The OPC Read block has the same name, update rate, and items as `GrpObj`. If all items in `GrpObj` have the same data type, the OPC Read block's `Value` port indicates that data type. `BlkPath` indicates the full path to the new OPC Read block.

`BlkPath = genslread(GrpObj,DestSys)` generates the OPC Read block and places it into the system defined by `DestSys`. `DestSys` must be a model name or a path to a subsystem block. The OPC Read block automatically takes a location that attempts to minimize overlap of lines and blocks, however, the block might appear over an existing annotation.

Examples

Create a group object with two items, and then construct an OPC Read block from the group.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
grp = addgroup(da,'ExOPCREAD');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-Toothed Waves.Int2');
% Set update rate to 2 seconds:
grp.UpdateRate = 2;
% Construct OPC Read block:
blkPath = genslread(grp)
```

Version History

Introduced before R2006a

See Also

Functions

`genslwrite`

genslwrite

Generate Simulink OPC Write block from MATLAB group object

Syntax

```
BlkPath = genslwrite(GrpObj)
BlkPath = genslwrite(GrpObj,DestSys)
```

Description

`BlkPath = genslwrite(GrpObj)` generates an OPC Write block from the `dagroup` object `GrpObj`, and places the block in a new Simulink model. The generated OPC Write block has the same name, update rate, and items as `GrpObj`. `BlkPath` indicates the full path to the new OPC Write block.

`BlkPath = genslwrite(GrpObj,DestSys)` generates the OPC Write block and places it into the system defined by `DestSys`. `DestSys` must be a model name or a path to a subsystem block. The OPC Write block automatically takes a location that attempts to minimize overlap of lines and blocks, however, the block might appear over an existing annotation.

Examples

Create a group object with two items, and then construct an OPC Write block from the group.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
grp = addgroup(da,'ExOPCREAD');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-Toothed Waves.Int2');
% Set update rate to 2 seconds:
grp.UpdateRate = 2;
% Construct OPC Write block:
blkPath = genslwrite(grp)
```

Version History

Introduced before R2006a

See Also

Functions
`genslread`

get

OPC object properties

Syntax

```
Val = get(Obj,'PropName')
get(Obj)
Val = get(Obj)
```

Description

`Val = get(Obj,'PropName')` returns the value `Val` of the property specified by the character vector or string `PropName`, for the OPC object `Obj`.

If `PropName` is an array of property names, `get` returns a 1-by-`N` cell array of values, where `N` is the length of `PropName`. If `Obj` is a vector of toolbox objects, `Val` is an `M`-by-`N` cell array of property values where `M` is equal to the length of `Obj` and `N` is equal to the number of properties requested.

`get(Obj)` displays all property names and their current values for the toolbox object `Obj`.

`Val = get(Obj)` returns a structure, `Val`, where each field name is the name of a property of `Obj` containing the value of that property. If `Obj` is an array of toolbox objects, `Val` is an `M`-by-1 structure array.

Examples

Obtain the values of the `Status` and `Group` properties of an `opcda` object, and then display all the properties of the object:

```
da = opcda('localhost','Dummy.Server');
get(da, {'Status','Group'})
out = get(da,'Status')
get(da)
```

Tips

As an alternative to the `get` function, you can directly retrieve property values using dot-notation. The following two lines achieve the same result.

```
t = get(daObj,'Timeout');
t = daObj.Timeout;
```

Version History

Introduced before R2006a

See Also

Functions
`opchelp` | `propinfo` | `set`

getAllChildren

Package: `opc.ua`

Recursively retrieve all children of OPC UA server node

Syntax

`AllChildNodes = getAllChildren(StartNode)`

Description

`AllChildNodes = getAllChildren(StartNode)` returns all children of a given node as a vector of Node objects, including all children recursively.

Note This function is memory intensive. Use it only when necessary. Alternatively, consider accessing the `Children` property of the node, or searching with `browseNamespace`, `findNodeByName`, or `findNodeById`.

Examples

This example shows how to return all children of the server node.

```
UaClient = opcua('localhost',51210);
connect(UaClient);
serverNode = UaClient.Namespace(1);
allServerNodes = getAllChildren(serverNode);
whos allServerNodes
```

```
  Name                Size              Bytes  Class          Attributes

  allServerNodes      1x349              2896  opc.ua.Node
```

Version History

Introduced in R2015b

See Also

`findNodeById` | `findNodeByName` | `browseNamespace` | `getNamespace`

getDescription

Package: `opc.hda`

Get description of OPC HDA aggregate type or item attribute

Syntax

```
DStr = getDescription(Obj,ID)
DStr = getDescription(Obj,NameStr)
```

Description

`DStr = getDescription(Obj,ID)` returns the description character vector associated with the aggregate type or item attribute given by `ID`. If `ID` is a vector, `DStr` is a cell array of description character vectors.

`DStr = getDescription(Obj,NameStr)` returns the description character vector associated with the aggregate type or item attribute given by the character vector or string `NameStr`. If `NameStr` is an array, `DStr` is a cell array of description character vectors.

Examples

Get a description of all aggregate types provided by the Matrikon Simulation Server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
allDesc = getDescription(hdaObj.Aggregates)
```

Get a description of all item attributes provided by the Matrikon Simulation Server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
allDesc = getDescription(hdaObj.ItemAttributes)
```

See Also

Functions
`getIDFromName`

getdata

Retrieve logged OPC records from toolbox engine to MATLAB workspace

Syntax

```
S = getdata(GObj)
S = getdata(GObj,NRec)
TSCell = getdata(GObj,'timeseries')
TSCell = getdata(GObj, NRec,'timeseries')
[ItmID,Val,Qual,TStamp,ETime] = getdata(GObj,'DataType')
[ItmID,Val,Qual,TStamp,ETime] = getdata(GObj,NRec,'DataType')
```

Description

`S = getdata(GObj)` returns the number of records specified in the `RecordsToAcquire` property of `dagroup` object `GObj`, from the toolbox software engine. `GObj` must be a scalar `dagroup` object.

`S` is an `NRec`-by-`1` structure array, where `NRec` is the number of records returned. `S` contains the fields `'LocalEventTime'` and `'Items'`. `LocalEventTime` is a date vector corresponding to the local event time for that record. `Items` is an `NItems`-by-`1` structure array containing the fields shown below.

Field Name	Description
`ItemID`	The fully qualified tag name, as a character vector.
`Value`	The data value. The data type is defined by the item's `DataType` property.
`Quality`	The data quality, as a character vector. For a description, see “OPC Quality” on page A-2.
`TimeStamp`	The time the value was changed, as a date vector.

`S = getdata(GObj,NRec)` retrieves the first `NRec` records from the toolbox engine.

`TSCell = getdata(GObj,'timeseries')` and
`TSCell = getdata(GObj, NRec,'timeseries')` assign the data received from the toolbox engine to a cell array of time series objects. `TSCell` contains as many time series objects as there are items in the group, with the name of each time series object set to the item ID. The quality value stored in the time series object is offset from the quality value returned by the OPC server by 128. The quality displayed by each is the same. Because each record logged might not contain information for every item, the time series objects have only as many data points as there are records containing information about that particular item ID.

`[ItmID,Val,Qual,TStamp,ETime] = getdata(GObj,'`*`DataType`*`')` and
`[ItmID,Val,Qual,TStamp,ETime] = getdata(GObj,NRec,'`*`DataType`*`')` assign the data retrieved from the toolbox engine to separate arrays. Valid data types are `'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'uint32'`, `'logical'`, `'currency'`, `'date'`, and `'cell'`.

`ItmID` is a `1`-by-`NItem` cell array of item names.

Val is an NRec-by-NItem array of values with the data type specified. If a data type of 'cell'is specified, then Val is a cell array containing data in the returned data type for each item. Otherwise, Val is a numeric array of the specified data type.

Note '*DataType*' must be set to 'cell' when retrieving records containing character vectors or arrays of values.

Qual is an NRec-by-NItem array of quality character vectors for each value in Val.

TStamp is an NRec-by-NItem array of MATLAB date numbers representing the time when the relevant value and quality were stored on the OPC server.

ETime is an NRec-by-1 array of MATLAB date numbers, corresponding to the local event time for each record.

Each record logged may not contain information for every item returned, since data for that item may not have changed from the previous update. When data is returned as a numeric matrix, the missing item columns for that record are filled as follows.

Argument	Behavior for Missing Items
Val	The corresponding value entry is set to the previous value of that item, or to NaN if there is no previous value.
Qual	The corresponding quality entry is set to 'Repeat'.
TStamp	The corresponding time stamp entry is set to the first valid time stamp for that record.

getdata is a blocking function that returns execution control to the MATLAB workspace when one of the following conditions is met:

- The requested number of records becomes available.
- The logging operation is automatically stopped by the engine. If fewer records are available than the number requested, a warning is generated and all available records are returned.
- You issue **Ctrl+C**. The logging task does not stop, and no data is removed from the toolbox engine.

When getdata completes, the object's RecordsAvailable property is reduced by the number of records returned by getdata.

Examples

Configure and start a logging task for 60 seconds of data.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExOPCREAD');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-Toothed Waves.Int2');
grp.LoggingMode = 'memory';
grp.RecordsToAcquire = 60;
start(grp);
```

Retrieve the first two records into a structure. This operation waits for at least two records.

```
s = getdata(grp,2)
```

Retrieve all the remaining data into a double array and plot it with a legend.

```
[itmID,val,qual,tStamp] = getdata(grp,'double');
plot(tStamp(:,1),val(:,1),tStamp(:,2),val(:,2));
legend(itmID);
datetick x keeplimits
```

Version History

Introduced before R2006a

See Also

Functions

`flushdata` | `peekdata` | `start` | `stop`

getIDFromName

Package: `opc.hda`

Translate OPC HDA aggregate type or item attribute name to numeric identifier

Syntax

```
ID = getIDFromName(Obj,NameStr)
```

Description

`ID = getIDFromName(Obj,NameStr)` returns the ID associated with the aggregate type or attribute item name `NameStr`. If `NameStr` is an array, `ID` is a vector of IDs.

Examples

Retrieve the ID of the `TIMEAVERAGE` item attribute provided by the Matrikon Simulation Server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
descID = getIDFromName(hdaObj.Aggregates,'TIMEAVERAGE')
```

Retrieve the ID of the `DESCRIPTION` item attribute provided by the Matrikon Simulation Server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
descID = getIDFromName(hdaObj.ItemAttributes,'DESCRIPTION')
```

See Also

Functions
`getDescription` | `getNameList`

getIDList

Package: `opc.hda`

Get all aggregate type or item attribute IDs

Syntax

`ID = getIDList(Obj)`

Description

`ID = getIDList(Obj)` returns all IDs stored in the OPC HDA aggregate type or item attribute object `Obj`.

Examples

Retrieve the IDs of the aggregate types provided by the Matrikon Simulation Server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
allIDs = getIDList(hdaObj.Aggregates)
```

Retrieve the IDs of the item attributes provided by the Matrikon Simulation Server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
allIDs = getIDList(hdaObj.ItemAttributes)
```

See Also

Functions
`getNameList`

getIndexFromID

Package: `opc.hda`

Indices matching OPC HDA data item IDs

Syntax

```
ind = getIndexFromID(dObj,'itemID')
ind = getIndexFromID(dObj,idCell)
```

Description

`ind = getIndexFromID(dObj,'itemID')` returns the index of HDA data object array `dObj` that matches the item ID `'itemID'`.

`ind = getIndexFromID(dObj,idCell)` returns the indices of HDA data object array `dObj` that match the item IDs contained in the cell array `idCell`. `idCell` must be a cell array of character vectors.

Examples

Load the OPC HDA example data file and find the index of `'Item Example.Item.2'`:

```
load opcSampleHdaData;
ind = getIndexFromID(hdaDataVis,'Example.Item.2');
```

getNameList

Package: `opc.hda`

Get all aggregate type or item attribute names

Syntax

```
NameCell = getNameList(Obj)
```

Description

`NameCell = getNameList(Obj)` returns all names stored in the OPC HDA aggregate type or item attribute object `Obj`. `NameCell` is a cell array of character vectors (even if `Obj` stores only one ID).

Examples

Retrieve the names of the aggregate types provided by the Matrikon Simulation Server.

```
hdaObj = opchda('localhost', 'Matrikon.OPC.Simulation');
connect(hdaObj);
allNames = getNameList(hdaObj.Aggregates)
```

Retrieve the names of the item attributes provided by the Matrikon Simulation Server.

```
hdaObj = opchda('localhost', 'Matrikon.OPC.Simulation');
connect(hdaObj);
allNames = getNameList(hdaObj.ItemAttributes)
```

See Also

`getIDList` | `getIDFromName`

getnamespace

OPC DA server name space

Syntax

```
S = getnamespace(DAObj)
S = getnamespace(DAObj,'Filter1',Val1,'Filter2',Val2, ...)
```

Description

`S = getnamespace(DAObj)` returns the entire name space of the server associated with the data access (`opcda`) object specified by `DAObj`. `S` is a recursive structure array representing the name space of the server. Each element of `S` is a node in the name space. `S` contains the fields:

- `Name` — a descriptive name
- `FullyQualifiedID` — the fully qualified `ItemID` of that node
- `NodeType` — defines the node as a `'branch'` node (containing other nodes) or `'leaf'` node (containing no other nodes)
- `Nodes` — a structure array with the same fields as `S`, representing the nodes contained in this branch of the name space.

Use `flatnamespace` to flatten the hierarchical name space.

`S = getnamespace(DAObj,'Filter1',Val1,'Filter2',Val2, ...)` allows you to filter the retrieved name space based on a number of available browse filters. Available filters are described in the table in Browse Filters on page 19-57.

Examples

Get Name Spaces

1 Get the entire name space for the Matrikon Simulation Server on the local host:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
nsFull = getnamespace(da)
```

2 Get only the first level of the name space:

```
nsPart = getnamespace(da,'Depth',1)
```

3 Add the nodes contained in the first branch of the name space to the existing structure:

```
nsPart(1).Nodes = getnamespace(da, ...
   'StartItemID', nsPart(1).FullyQualifiedID, ...
   'Depth',1);
```

Browse Filters

BrowseFilter	Description
`'StartItemID'`	Specify the `FullyQualifiedID` of a branch node, as a character vector or string. Only nodes contained in that branch node will be returned. Some OPC servers do not support partial name space retrieval based on this option: An error is generated if you attempt to use the `'StartItemID'` browse filter on such a server.
`'Depth'`	Specify the depth of the name space that you want returned. A `'Depth'` value of `1` returns only the nodes contained in the starting position. A `'Depth'` value of `2` returns the nodes contained in the starting position and all of their nodes. A `'Depth'` value of `Inf` returns all nodes. When combined with the `'StartItemID'` filter, the `'Depth'` filter provides a useful way to investigate a name server hierarchy one layer at a time.
`'AccessRights'`	Restricts the search to leaf nodes with particular access right characteristics. Specify `'read'` to return nodes that include the read access right, and `'write'` to return nodes that include the write access right. An empty character vector (`''`) returns nodes with any access rights. Note that branch nodes will still be returned in the name space, in order to contain the leaf nodes that have the requested access rights.
`'DataType'`	Restricts the search to nodes with a particular canonical data type. Valid data types are `'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'uint32'`, `'logical'`, `'currency'`, and `'date'`. Use the `'DataType'` filter to find server items with a specific data type, such as `'double'` or `'date'`. Note that branch nodes will still be returned in the name space, in order to contain the leaf nodes that have the required data type.

Version History
Introduced before R2006a

See Also

Functions
`additem` | `flatnamespace` | `serveritems`

getNameSpace

Package: `opc.hda`

OPC HDA server name space

Syntax

```
NS = getNameSpace(HdaObj)
NS = getNameSpace(HdaObj,'StartItemID','itemID')
NS = getNameSpace(HdaObj,'Depth',dLevel)
NS = getNameSpace(HdaObj,'StartItemID','itemID','Depth',dLevel)
```

Description

`NS = getNameSpace(HdaObj)` retrieves the entire server name space from the connected OPC HDA Client `HdaObj`.

`NS = getNameSpace(HdaObj,'StartItemID','itemID')` retrieves the server name space beginning at Fully Qualified Item ID `'itemID'`, and all branches in the name space below `'itemID'`.

`NS = getNameSpace(HdaObj,'Depth',dLevel)` retrieves the `dLevel` levels of the server name space beginning at the server name space root. Specifying `dLevel` as `1` retrieves only the nodes (branch and leaf) contained in the root of the server name space.

`NS = getNameSpace(HdaObj,'StartItemID','itemID','Depth',dLevel)` retrieves the `dLevel` levels of the name space starting at Fully Qualified Item ID `'itemID'`.

In all cases, `NS` is a recursive structure array representing the name space of the server. Each element of `NS` is a node in the name space. `NS` contains the fields:

- `Name` — a descriptive name
- `FullyQualifiedID` — the fully qualified `ItemID` of that node
- `NodeType` — defines the node as a `'branch'` node (containing other nodes) or `'leaf'` node (containing no other nodes)
- `Nodes` — a structure array with the same fields as `NS`, representing the nodes contained in this branch of the name space

Use `flatnamespace` to flatten the hierarchical name space.

Examples

Get Name Spaces

1 Get the entire name space for the Matrikon Simulation Server on the local host:

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
nsFull = getNameSpace(hdaObj)
```

2 Get only the first level of the name space:

```
nsPart = getNameSpace(hdaObj,'Depth',1)
```

3 Add the nodes contained in the first branch of the name space to the existing structure:

```
nsPart(1).Nodes = getNameSpace(hdaObj, ...
    'StartItemID',nsPart(1).FullyQualifiedID, ...
    'Depth',1);
```

Version History

Introduced in R2011a

See Also

Functions

`connect`

getNamespace

Package: `opc.ua`

Namespace of OPC UA server associated with client

Syntax

```
nodes = getNamespace(UaClient)
nodes = getNamespace(UaClient,BrowseNode)
nodes = getNamespace( ___ ,'-force')
```

Description

`nodes = getNamespace(UaClient)` retrieves one layer of the namespace of the server associated with client object `UaClient`. The namespace is stored in the `Namespace` property of `uaClient` as a hierarchical tree of nodes.

`nodes = getNamespace(UaClient,BrowseNode)` retrieves only the nodes referenced from `BrowseNode`, and stores them in the `Children` property of `BrowseNode`. If the `BrowseNode` argument is empty or omitted, the first layer of the namespace is retrieved and stored in the client.

`getNamespace` might not need to retrieve nodes from the server. If the nodes already exist locally, they are returned automatically.

`nodes = getNamespace( ___ ,'-force')` forces retrieval of the `Children` property contents from the server again and stores them in `BrowseNode`, even if the nodes already exist locally.

Note When retrieving a namespace with many children, you should allow a significant amount of time for this function to complete, especially when displaying the results in the MATLAB command window. For example, retrieving a namespace with several thousand children could take up to a minute or more.

Examples

Retrieve One Layer of Namespace

This example shows how to retrieve one layer of the namespace from the OPC UA client.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
nodes = getNamespace(UaClient)
```

```
nodes =
1x4 OPC UA Node array:
    index      Name        NsInd  Identifier  NodeType  Children
    -----  -------------  -----  ----------  --------  --------
      1    Server         0      2253        Object    10
    2    Data           2      10157       Object    3
    3    Boilers        4      1240        Object    2
    4    MemoryBuffers  7      1025        Object    2
```

Input Arguments

`UaClient` — OPC UA client
OPC UA client object

OPC UA client, specified as an OPC UA client object

`BrowseNode` — Browse node
node object

Browse node, specified as a node object.

Output Arguments

`nodes` — Layer of namespace tree from server
structure

Layer of namespace tree from server, returned as a structure.

Version History

Introduced in R2015b

See Also

Functions
`browseNamespace`

getNodeAttributes

Package: `opc.ua`

Read OPC UA server node attributes

Syntax

```
Values = getNodeAttributes(UaClient,NodeList,AttributeIds)
Values = getNodeAttributes(NodeList,AttributeIds)
```

Description

`Values = getNodeAttributes(UaClient,NodeList,AttributeIds)` reads from the server the attributes defined by `AttributeIds` for the nodes identified by `NodeList`. You can define node objects for `NodeList` using `getNamespace` or `browseNamespace`.

`Values = getNodeAttributes(NodeList,AttributeIds)` reads from the nodes identified by `NodeList`. All nodes must be of the same connected client.

Examples

Read node attributes

This example shows how to read node attributes from the server for one layer of the namespace.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
NodeList = getNamespace(UaClient);
Values = getNodeAttributes(UaClient,NodeList,{'NodeId','Description'})
```

```
Values =
4x1 struct array with fields:
    NodeId
    Description
```

Input Arguments

`UaClient` — OPC UA client
OPC UA client object

OPC UA client, specified as an OPC UA client object

`NodeList` — List of nodes
array of node objects

List of nodes, specified as an array of node objects. For information on node object functions and properties, type

```
help opc.ua.Node
```

`AttributeIds` — Server attributes
array of uint32, cell array of character vectors, or array of strings

Server attributes specified as an array of uint32, cell array of character vectors, or array of strings. For information on server `AttributeId` values, type

```
help opc.ua.AttributeId
```

Output Arguments

`Values` — Attribute values
structure

Attribute values, returned as a structure. The structure array contains the fields given by the `AttributeIds`. If an attribute cannot be read for a node, the relevant field will be empty.

Version History

Introduced in R2015b

See Also

Functions
`getNamespace` | `browseNamespace` | `readValue`

getServerStatus

Package: `opc.ua`

Status of OPC UA server

Syntax

```
sstat = getServerStatus(UaClient)
```

Description

`sstat = getServerStatus(UaClient)` retrieves the status of the OPC UA server connected to `UaClient`. `UaClient` must be a scalar connected OPC UA client, not a vector of clients.

`sstat` is returned as a structure containing the following fields:

Field name	Description
`StartTime`	Time the server started (MATLAB datetime)
`CurrentTime`	Current time on the server (MATLAB datetime)
`State`	State of the server (character vector)
`BuildInfo`	Structure describing the build information for the server, including `ManufacturerName`, `ProductName`, and `SoftwareVersion`
`SecondsTillShutdown`	If the server is shutting down, how long until shutdown occurs
`ShutdownReason`	Reason for the server shutdown, or an empty character vector

Examples

Connect an OPC UA client and retrieve the status of its server.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
sstat = getServerStatus(UaClient)
```

```
sstat =
              StartTime: 10-Jun-2015 16:39:17
            CurrentTime: 10-Jun-2015 16:55:00
                  State: 'Running'
              BuildInfo: [1x1 struct]
    SecondsTillShutdown: 0
         ShutdownReason: ''
```

Version History

Introduced in R2015b

See Also

`connect` | `disconnect` | `opcua` | `opcuaserverinfo`

int16

Package: `opc.hda`

Convert OPC HDA data object array to int16 matrix

Syntax

`Vint16 = int16(DObj)`

Description

`Vint16 = int16(DObj)` converts the OPC HDA data object array `DObj` into an `int16` matrix.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to int16 Matrix

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create an `int16` matrix from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vint16 = int16(dUnion);
```

Input Arguments

`DObj` — OPC HDA data

OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vint16` — OPC HDA data values

`int16` matrix

OPC HDA data values, returned as an `int16` matrix. `Vint16` is constructed as an `M`-by-`N` matrix of `int16` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

int32

Package: `opc.hda`

Convert OPC HDA data object array to int32 matrix

Syntax

`Vint32 = int32(DObj)`

Description

`Vint32 = int32(DObj)` converts the OPC HDA data object array `DObj` into an `int32` matrix.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to int32 Matrix

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create an `int32` matrix from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vint32 = int32(dUnion);
```

Input Arguments

`DObj` — OPC HDA data
OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vint32` — OPC HDA data values
`int32` matrix

OPC HDA data values, returned as an `int32` matrix. `Vint32` is constructed as an `M`-by-`N` matrix of `int32` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

int64

Package: `opc.hda`

Convert OPC HDA data object array to int64 matrix

Syntax

`Vint64 = int64(DObj)`

Description

`Vint64 = int64(DObj)` converts the OPC HDA data object array `DObj` into an `int64` matrix.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to int64 Matrix

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create an `int64` matrix from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vint64 = int64(dUnion);
```

Input Arguments

`DObj` — OPC HDA data
OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vint64` — OPC HDA data values
`int64` matrix

OPC HDA data values, returned as an `int64` matrix. `Vint64` is constructed as an `M`-by-`N` matrix of `int64` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

int8

Package: `opc.hda`

Convert OPC HDA data object array to int8 matrix

Syntax

`Vint8 = int8(DObj)`

Description

`Vint8 = int8(DObj)` converts the OPC HDA data object array `DObj` into an `int8` matrix.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to int8 Matrix

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create an `int8` matrix from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vint8 = int8(dUnion);
```

Input Arguments

`DObj` — OPC HDA data
OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vint8` — OPC HDA data values
`int8` matrix

OPC HDA data values, returned as an `int8` matrix. `Vint8` is constructed as an `M`-by-`N` matrix of `int8` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

isConnected

Package: `opc.hda`

True if HDA Client is connected to server

Syntax

```
isConnected(hdaObj)
```

Description

`isConnected(hdaObj)` returns `true` if the OPC HDA Client object `hdaObj` is connected to an OPC HDA server, and `false` otherwise.

If `hdaObj` is an array, `isConnected` returns an array the same size as `hdaObj`, containing `true` where that respective element of `hdaObj` is connected to a server and `false` otherwise.

Examples

Create an HDA client for the Matrikon Simulation Server and connect to the server:

```
hdaObj = opchda('localhost', 'Matrikon.OPC.Simulation');
connect(hdaObj);
```

Check the status of the connection:

```
tf = isConnected(hdaObj)
```

Version History

Introduced in R2011a

See Also

`connect` | `disconnect`

isConnected

Package: `opc.ua`

Determine if OPC UA client object is connected to server

Syntax

`tf = isConnected(UaClient)`

Description

`tf = isConnected(UaClient)` returns true (logical 1) if the client `UaClient` is connected to the server, or false (logical 0) otherwise. If `UaClient` is a vector of client objects, `tf` is a vector representing the connected state of each client.

Examples

Connect an OPC UA client and view its connection status.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
isConnected(UaClient)
```

```
1
```

```
disconnect(UaClient);
isConnected(UaClient)
```

```
0
```

Version History

Introduced in R2015b

See Also

`connect` | `disconnect`

isEmptyNode

Package: `opc.ua`

True for empty nodes of OPC UA node array

Syntax

```
tf = isEmptyNode(NodeObj)
```

Description

`tf = isEmptyNode(NodeObj)` returns true (logical 1) for nodes that are empty, or false (logical 0) otherwise. A node is empty if its `NamespaceIndex` or `Identifier` property is empty. You cannot use an empty node in any read, write, or query operation on a connected client.

Examples

Determine If Nodes Are Empty

Identify which nodes in an array are empty.

Browse the namespace to select nodes. This example selects two.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
nodes = browseNamespace(UaClient)

nodes =

1x2 OPC UA Node array:
    index   Name   NsInd  Identifier  NodeType  Children
    -----  ------  -----  ----------  --------  --------
      1    FTX001  4      1243        Object    1
      2    Output  4      1244        Variable  1
```

You can create an array of nodes that contains results from separate browsing results. Assign the latest nodes to this array, and verify which nodes are empty.

```
nodeArray(3:4) = nodes;
tf = isEmptyNode(nodeArray)

tf =

  1x4 logical array

   1   1   0   0
```

The result indicates that elements 1 and 2 are empty nodes.

Input Arguments

`NodeObj` — OPC UA nodes
array of node objects

OPC UA nodes, specified as an array of node objects.

Example: `NodeObj = opcuanode()`

Output Arguments

`tf` — Indication that node is empty
true (1) | false (0)

Indication that node is empty, returned as a logical value or array of logical values. A value of `true` (1) indicates an empty node.

Version History

Introduced in R2016b

See Also

Functions
`isVariableType` | `isObjectType` | `opcuanode`

isObjectType

Package: `opc.ua`

True for object nodes of OPC UA server

Syntax

```
tf = isObjectType(NodeObj)
```

Description

`tf = isObjectType(NodeObj)` returns true (logical 1) for nodes that are object type nodes, or false (logical 0) otherwise. You cannot read current and historical values from object type nodes. Object nodes are used to organize the server name space.

Examples

Identify some nodes and determine if they are object type nodes.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
nodes = browseNamespace

nodes =
1x2 OPC UA Node array:
    index     Name      NsInd  Identifier  NodeType  Children
    -----  -----------  -----  ----------  --------  --------
      1    DoubleValue  2      10226       Variable  0
      2    Scalar       2      10159       Object    29

isObjectType(nodes(1))

ans =
     0

isObjectType(nodes(2))

ans =
     1
```

Version History

Introduced in R2015b

See Also

`isVariableType` | `opcuanode`

isvalid

True for undeleted OPC objects

Syntax

```
A = isvalid(Obj)
```

Description

`A = isvalid(Obj)` returns a logical array, `A`, that contains `false` where the elements of `Obj` are deleted OPC objects and `true` where the elements of `Obj` are valid objects.

Use the `clear` command to clear an invalid toolbox object from the workspace.

Examples

Create two valid OPC data access objects, and then delete one to make it invalid:

```
da(1) = opcda('localhost','Dummy.ServerA');
da(2) = opcda('localhost','Dummy.ServerB');
out1 = isvalid(da)
% Delete the first object and show it is invalid:
delete(da(1))
out2 = isvalid(da)
% Delete the second object and clear the object array:
clear da
```

Version History

Introduced before R2006a

See Also

`delete` | `opchelp`

isVariableType

Package: opc.ua

True for variable nodes of OPC UA server

Syntax

```
tf = isVariableType(NodeObj)
```

Description

`tf = isVariableType(NodeObj)` returns true (logical 1) for nodes that are variable type nodes, or false (logical 0) otherwise. You can write values, and read current and historical values from variable type nodes.

Examples

Identify some node and determine if they are variable type nodes.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
connect(UaClient);
nodes = browseNamespace(UaClient)

nodes =
1x2 OPC UA Node array:
    index     Name      NsInd  Identifier  NodeType  Children
    -----  -----------  -----  ----------  --------  --------
      1    DoubleValue  2      10226       Variable  0
      2    Scalar       2      10159       Object    29

isVariableType(nodes(1))

ans =
     1

isVariableType(nodes(2))

ans =
     0
```

Version History

Introduced in R2015b

See Also

Functions

`isObjectType` | `opcuanode`

load

Load OPC objects from MAT-file

Syntax

```
load FileName
load FileName Obj1 Obj2 ...
S = load('FileName','Obj1','Obj2',...)
```

Description

`load FileName` returns all variables from the MAT-file `FileName` into the MATLAB workspace.

`load FileName Obj1 Obj2 ...` returns the specified OPC objects, `Obj1`, `Obj2`, `...` from the MAT-file `FileName` into the MATLAB workspace.

`S = load('FileName','Obj1','Obj2',...)` returns the structure `S` with the specified toolbox objects `Obj1`, `Obj2`, `...` from the MAT-file `FileName`, instead of directly loading the toolbox objects into the workspace. The field names in `S` match the names of the retrieved toolbox objects. If you specify no objects, `load` returns all variables from the MAT-file.

When you load an object, its read-only properties initially take their default values. For example, the Status property value of an `opcda` object is `'disconnected'`. Use `propinfo` to determine if a property is read-only.

Examples

Assume the example on the `save` reference page saved the group object `grp` in the file `mygroup`. Load the group object from `mygroup`, and create a reference to the parent client.

```
load mygroup
da = grp.Parent;
```

Version History

Introduced before R2006a

See Also

Functions
`opchelp` | `propinfo` | `save`

logical

Package: `opc.hda`

Convert OPC HDA data object array to logical matrix

Syntax

`Vlogical = logical(DObj)`

Description

`Vlogical = logical(DObj)` converts the OPC HDA data object array `DObj` into a matrix of data type `logical`.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to Matrix of logicals

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create a matrix of type `logical` from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vlogical = logical(dUnion);
```

Input Arguments

`DObj` — OPC HDA data
OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vlogical` — OPC HDA data values
matrix of `logical` type

OPC HDA data values, returned as a matrix of `logical` type. `Vlogical` is constructed as an `M`-by-`N` matrix of `logical` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

makepublic

Convert private group into public group

Syntax

```
makepublic(GObj)
```

Description

`makepublic(GObj)` makes the `dagroup` object `GObj` public. Public groups allow you to share data configuration information across multiple OPC clients. Use the `GroupType` property to check whether a group is public.

Public groups on a server cannot have the same name. If you attempt to call `makepublic` on a private group with the same name as an existing public group, you get an error.

After you make a group public, you cannot add items to that group or delete items from that group. You must ensure that a group contains the required items before making the group public.

Not all OPC data access servers support public groups. If you try to make a public group on a server that does not support public groups, you get an error. To verify that a server supports public groups, use the `opcserverinfo` function on the client connected to that server: Look for an entry `'IOPCPublicGroups'` in the `'SupportedInterfaces'` field.

Use the `clonegroup` function to create a private group from a public group.

Examples

Create a group on a local server and make the group public:

```
da = opcda('localhost', 'Dummy.Server');
connect(da);
grp = addgroup(da, 'MakepublicEx');
itm1 = additem(grp, 'Device1.Item1');
itm2 = additem(grp, 'Device1.Item2');
makepublic(grp);
```

Version History

Introduced before R2006a

See Also

`clonegroup` | `opcserverinfo`

maskWrite

Perform mask write operation on a holding register

Syntax

```
maskWrite(m,address,andMask,orMask)
maskWrite(m,address,andMask,orMask,serverId)
```

Description

`maskWrite(m,address,andMask,orMask)` writes data to Modbus object `m` to a holding register at address `address`, using the indicated mask values. The function can set or clear individual bits in a specific holding register. It is a read/modify/write operation, and uses a combination of an AND mask, an OR mask, and the current contents of the register.

`maskWrite(m,address,andMask,orMask,serverId)` additionally specifies the `serverId` as the address of the server to send the write command to.

Examples

Perform a Mask Read on a Holding Register

You can modify the contents of a holding register using the `maskWrite` function. The function can set or clear individual bits in a specific holding register. It is a read/modify/write operation, and uses a combination of an AND mask, an OR mask, and the current contents of the register.

Create the `AND` and `OR` variables.

```
andMask = 6
orMask = 0
```

Set bit 0 at address 20, and perform a mask write operation. Since the `andMask` is a 6, that clears all bits except for bits 1 and 2. Bits 1 and 2 are preserved.

```
maskWrite(m,20,andMask,orMask)
```

Perform a Mask Read on a Holding Register, and Specify Server ID

Use the `serverId` argument to specify the address of the server to send the mask write command to.

Set bit 0 at address 20 and perform a mask write operation at server ID 3.

```
maskWrite(m,20,6,0,3)
```

Input Arguments

`address` — Register address to perform mask write operation on
double

Register address to perform mask write operation on, specified as a double. Address must be the first argument after the object name. This example sets bit 0 at address 20 and performs a mask write operation.

Example: `maskWrite(m,20,andMask,orMask)`

Data Types: `double`

`andMask` — AND value to use in mask write operation
double

AND value to use in mask write operation, specified as a double. `andMask` must be the second argument after the object name. The valid range is `0–65535`.

This example sets bit 0 at address 20 and performs a mask write operation, using 6 as the AND value.

Example: `maskWrite(m,20,6,0)`

Data Types: `double`

`orMask` — OR value to use in mask write operation
double

OR value to use in mask write operation, specified as a double. `orMask` must be the third argument after the object name. The valid range is `0–65535`.

This example sets bit 0 at address 20 and performs a mask write operation, using 0 as the OR value.

Example: `maskWrite(m,20,6,0)`

Data Types: `double`

`serverId` — Address of the server to send the mask write command to
double

Address of the server to send the mask write command to, specified as a double. Server ID must be specified after the object name, address, AND mask, and OR mask. If you do not specify a `serverId`, the default of `1` is used. Valid values are `0–247`, with `0` being the broadcast address. This example sets bit 0 at address 20 and performs a mask write operation at server ID 3.

Example: `maskWrite(m,20,6,0,3)`

Data Types: `double`

Tips

The function algorithm works as follows:

```
 Result = (register value AND andMask) OR (orMask AND (NOT andMask))
```

For example:

```
                 Hex    Binary
Current contents  12   0001 0010
And_Mask          F2   1111 0010
Or_Mask           25   0010 0101
(NOT And_Mask)    0D   0000 1101

Result            17   0001 0111
```

If the `orMask` value is 0, the result is simply the logical ANDing of the current contents and the `andMask`. If the `andMask` value is 0, the result is equal to the `orMask` value.

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`modbus` | `read` | `write` | `writeRead`

Topics
“Create a Modbus Connection” on page 18-3
“Configure Properties for Modbus Communication” on page 18-5
“Modify the Contents of a Holding Register Using a Mask Write” on page 18-18

modbus

Create Modbus object

Syntax

```
m = modbus(Transport,DeviceAddress)
m = modbus(Transport,DeviceAddress,Port)
m = modbus(Transport,DeviceAddress,Name,Value)
m = modbus(Transport,'Port')
m = modbus(Transport,'Port',Name,Value)
```

Description

Note In R2022a, Modbus functionality has moved from Instrument Control Toolbox™ to Industrial Communication Toolbox. If you have been a licensed Instrument Control Toolbox user prior to this release, you might be eligible to continue using Modbus functionality as described in Opt-In Offer for Instrument Control Toolbox Modbus Users.

`m = modbus(Transport,DeviceAddress)` constructs a Modbus object, `m`, over the transport type `Transport` using the specified `'DeviceAddress'`. When the transport is `'tcpip'`, `DeviceAddress` must be specified as the second argument. `DeviceAddress` is the IP address or host name of the Modbus server.

`m = modbus(Transport,DeviceAddress,Port)` additionally specifies `Port`. When the transport is `'tcpip'`, `DeviceAddress` must be specified. `Port` is the remote port used by the Modbus server. Port is optional, and it defaults to 502, which is the reserved port for Modbus.

`m = modbus(Transport,DeviceAddress,Name,Value)` specifies additional options with one or more name-value pair arguments using any of the previous syntaxes. For example, you can specify a timeout value. The `Timeout` property specifies the waiting time to complete read and write operations in seconds, and the default is `10`.

`m = modbus(Transport,'Port')` constructs a Modbus object `m` over the transport type `Transport` using the specified `'Port'`. When the transport is `'serialrtu'`, `'Port'` must be specified. This argument is the serial port the Modbus server is connected to, such as `'COM3'`.

`m = modbus(Transport,'Port',Name,Value)` specifies additional options with one or more name-value pair arguments using any of the previous syntaxes. For example, you can specify `NumRetries`, the number of retries to perform if there is no reply from the server after a timeout.

Examples

Create Object Using TCP/IP Transport

When the transport is TCP/IP, you must specify the IP address or host name of the Modbus server. You can optionally specify the remote port used by the Modbus server. Port defaults to 502, which is the reserved port for Modbus.

Create the Modbus object m using the host address shown and port of 308.

```
m = modbus('tcpip', '192.168.2.1', 308)
```

```
m = 

   Modbus TCPIP with properties:

    DeviceAddress: '192.168.2.1'
             Port: 308
           Status: 'open'
       NumRetries: 1
          Timeout: 10 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

The object output shows both the arguments you set and the defaults.

Create Object Using Serial RTU Transport

When the transport is 'serialrtu', you must specify 'Port'. This is the serial port the Modbus server is connected to.

Create the Modbus object m using the Port of 'COM3'.

```
m = modbus('serialrtu','COM3')
```

```
m = 

Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 10 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

The object output shows arguments you set and defaults that are used automatically.

Create Object and Set a Property

You can create the object using a name-value pair to set the properties such as Timeout. The Timeout property specifies the maximum time in seconds to wait for a response from the Modbus server, and the default is 10. You can change the value either during object creation or after you create the object.

For the list and description of properties you can set for both transport types, see “Configure Properties for Modbus Communication” on page 18-5.

Create a Modbus object using Serial RTU, but increase the `Timeout` to `20` seconds.

```
m = modbus('serialrtu','COM3','Timeout',20)
```

```
m = 

Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 20 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

The object output reflects the `Timeout` property change.

Input Arguments

`Transport` — Physical transport layer for device communication
character vector | string

Physical transport layer for device communication, specified as a character vector or string. Specify transport type as the first argument when you create the `modbus` object. You must set the transport type as either `'tcpip'` or `'serialrtu'` to designate the protocol you want to use.

Example: `m = modbus('tcpip','192.168.2.1')`

Data Types: `char`

`DeviceAddress` — IP address or host name of Modbus server
character vector | string

IP address or host name of Modbus server, specified as a character vector or string. If transport is TCP/IP, it is required as the second argument during object creation.

Example: `m = modbus('tcpip','192.168.2.1')`

Data Types: `char`

`Port` — Remote port used by Modbus server
502 (default) | double

Remote port used by Modbus server, specified as a double. Optional as a third argument during object creation if transport is TCP/IP. The default of 502 is used if none is specified.

Example: `m = modbus('tcpip','192.168.2.1',308)`

Data Types: `double`

`'Port'` — Serial port Modbus server is connected to
character vector | string

Serial port Modbus server is connected to, e.g. `'COM1'`, specified as a character vector or string. If transport is Serial RTU, it is required as the second argument during object creation.

Example: `m = modbus('serialrtu','COM3')`

Data Types: `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

There are a number of name-value pairs that can be used when you create the `modbus` object, including the two shown here. Some can only be used with either TCP/IP or Serial RTU, and some can be used with both transport types. For a list of all the properties and how to set them both during and after object creation, see “Configure Properties for Modbus Communication” on page 18-5.

Example: `m = modbus('serialrtu','COM3','Timeout',20)`

`Timeout` — Maximum time in seconds to wait for a response from the Modbus server
10 (default) | double

Maximum time in seconds to wait for a response from the Modbus server, specified as the comma-separated pair consisting of `'Timeout'` and a positive value of type `double`. The default is `10`. You can change the value either during object creation or after you create the object.

Example: `m = modbus('serialrtu','COM3','Timeout',20)`

Data Types: `double`

`NumRetries` — Number of retries to perform if there is no reply from the server after a timeout
double

Number of retries to perform if there is no reply from the server after a timeout, specified as the comma-separated pair consisting of `'NumRetries'` and a positive value of type `double`. If using the Serial RTU transport, the message is resent. If using the TCP/IP transport, the connection is closed and reopened. You can change the value either during object creation, or after you create the object.

Example: `m = modbus('serialrtu','COM3','NumRetries',5)`

Data Types: `double`

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`read` | `write` | `writeRead` | `maskWrite`

Topics
“Create a Modbus Connection” on page 18-3
“Configure Properties for Modbus Communication” on page 18-5

Modbus Explorer

Read and write to Modbus coils and registers

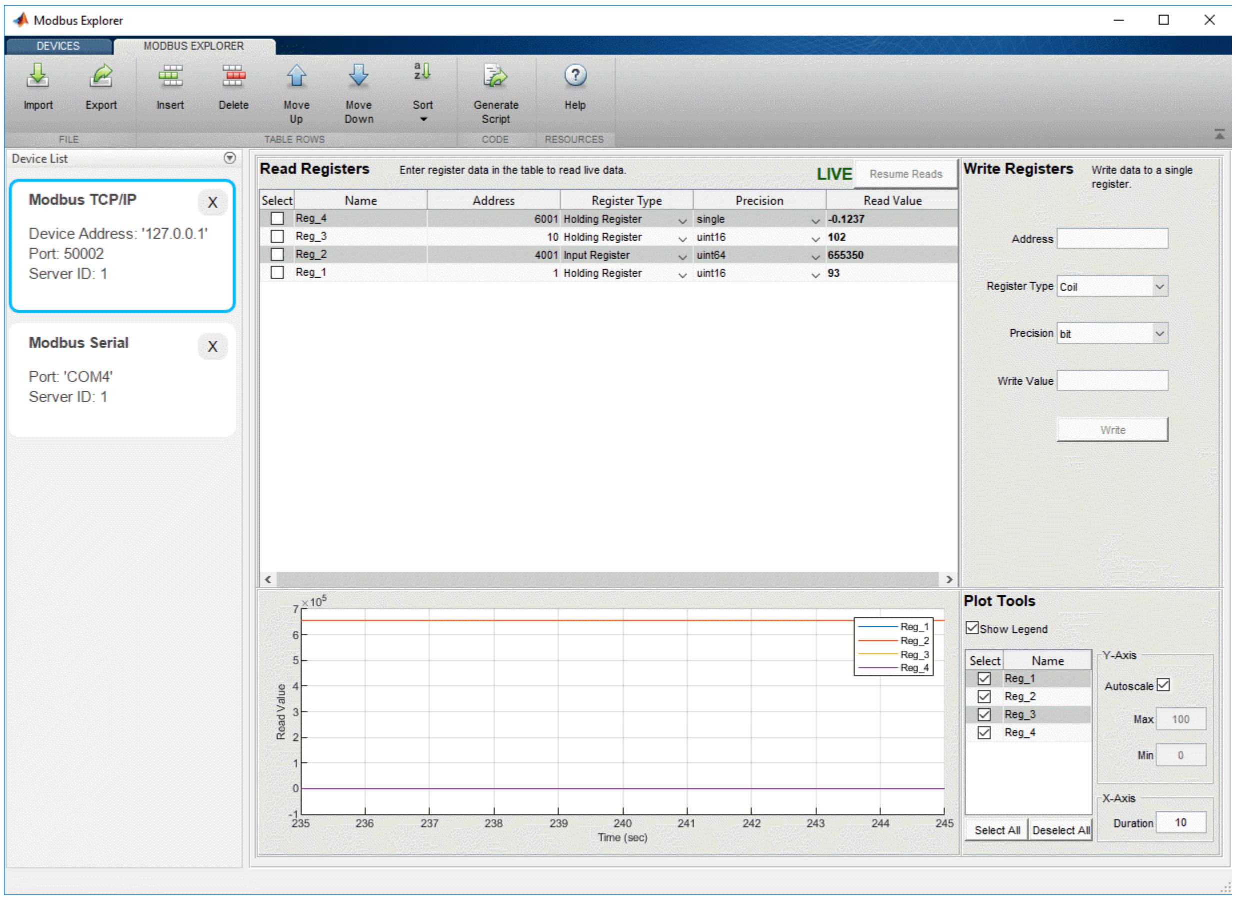
Description

The **Modbus Explorer** app enables you to read and write to registers through Instrument Control Toolbox without having to write a MATLAB script.

The **Modbus Explorer** app offers a user interface to easily set up read and write operations, and a live plot to see the values. The read table allows you to easily organize and manage reads for multiple addresses, such as different sensors and switches on a PLC. The app supports a subset of the MATLAB MATLAB functionality. You can do the following in the **Modbus Explorer** app:

- Read coils, inputs, input registers, and holding registers. This is the functionality of the Modbus `read` function.
- Write to coils and holding registers. This is the functionality of the Modbus `write` function.

The app does not support the functionality of the Modbus `writeRead` function or the `maskWrite` function.



Modbus TCP/IP
Device Address: '127.0.0.1'
Port: 50002
Server ID: 1

Modbus Serial
Port: 'COM4'
Server ID: 1

Read Registers — Enter register data in the table to read live data.

Select	Name	Address	Register Type	Precision	Read Value
☐	Reg_4	6001	Holding Register	single	-0.1237
☐	Reg_3	10	Holding Register	uint16	102
☐	Reg_2	4001	Input Register	uint64	655350
☐	Reg_1	1	Holding Register	uint16	93

Write Registers — Write data to a single register.

Address:
Register Type: Coil
Precision: bit
Write Value:

☑ Show Legend

Select	Name
☑	Reg_1
☑	Reg_2
☑	Reg_3
☑	Reg_4

Y-Axis: Autoscale ☑ Max 100 Min 0
X-Axis: Duration 10

Open the Modbus Explorer App

- MATLAB Toolstrip: On the **Apps** tab, under **Test & Measurement**, click the app icon.
- MATLAB command prompt: Enter `modbusExplorer`.

Examples

- “Configure a Connection in the Modbus Explorer” on page 18-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 18-26
- “Control a PLC Using the Modbus Explorer” on page 18-28

Version History

Introduced in R2019a

Topics

“Configure a Connection in the Modbus Explorer” on page 18-20
“Read Coils, Inputs, and Registers in the Modbus Explorer” on page 18-23
“Write to Coils and Holding Registers in the Modbus Explorer” on page 18-26
“Control a PLC Using the Modbus Explorer” on page 18-28

mqttclient

Create MQTT client connected to broker

Description

An `icomm.mqtt.Client` object represents an MQTT client in MATLAB that connects to an external MQTT broker.

Creation

Syntax

```
mqttClient = mqttclient(brokerAddr)
mqttClient = mqttclient(brokerAddr,Name=Value)
```

Description

`mqttClient = mqttclient(brokerAddr)` creates an MQTT client connected to the broker specified by `brokerAddr`. `brokerAddr` is a host name or IP address of the MQTT broker including the connection protocol. Supported protocols include TCP, WS, SSL, and WSS.

`mqttClient = mqttclient(brokerAddr,Name=Value)` specifies function options and properties of `mqttClient` using optional name-value pairs.

Input Arguments

`brokerAddr` — Location of MQTT broker
string | char

Location of MQTT broker as a URL with protocol, specified as a string or character vector.

Example: `"tcp://broker.hivemq.com"`

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Port=8883`

Name-value arguments can specify the properties `Port`, `ClientID`, `Timeout`, and `KeepAliveDuration`; and the following options:

`Username` — User name for connection to broker
string | char

User name for connection to broker, specified as a string or character vector.

Data Types: `char` | `string`

`Password` — User password for connection to broker
string | char

User password for connection to broker, specified as a string or character vector.

Data Types: `char` | `string`

`CARootCertificate` — Server root certificate for broker authentication
string | char

Server root certificate for broker authentication during a secure connection, specified as a string or character vector.

Data Types: `char` | `string`

`ClientCertificate` — Certificate for client authentication
string | char

Certificate for client authentication during a secure connection, specified as a string or character vector.

Data Types: `char` | `string`

`ClientKey` — Private key file for client authentication
string | char

Private key file for client authentication, used along with `ClientCertificate` for authentication during secure connection.

Data Types: `char` | `string`

`SSLPassword` — Password to decrypt private key
string | char

Password to decrypt the private `ClientKey` file, specified as a string or character vector.

Data Types: `char` | `string`

Properties

`Port` — Socket port number for connection
integer value

This property is read-only.

Socket port number to use when connecting to the MQTT broker, specified as an integer value.

Example: 8883

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`ClientID` — Identifier of client
string | char

This property is read-only.

Identifier of client for connection to broker, specified as a string or character vector.

Data Types: `char` | `string`

`Timeout` — Time allowed to complete connection
5 (default) | integer value | duration

This property is read-only.

Time allowed for connection to be completed, specified as a numeric integer value in seconds or as a duration.

Example: `Timeout=60`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`KeepAliveDuration` — Maximum idle time allowed between broker and client
60 (default) | integer value | duration

This property is read-only.

Maximum idle time allowed between broker and client, specified as a numeric integer value in seconds or as a duration. If no traffic occurs in this time span, the client issues a keep alive packet.

Example: `KeepAliveDuration=minutes(5)`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `duration`

`BrokerAddress` — Location of MQTT broker
string | char

This property is read-only.

Location of the MQTT broker, specified as a string or character vector. BrokerAddress identifies the host name or IP address of the MQTT broker, including connection protocol. Supported protocols include TCP, WS, SSL, and WSS.

Example: `"tcp://broker.hivemq.com"`

Data Types: `char` | `string`

`Subscriptions` — Table of topic subscriptions
table

This property is read-only.

Table of topics client is subscribed to.

Data Types: `table`

`Connected` — Status of client connection to broker
`1` | `0`

This property is read-only.

Status of the client connection to the broker, returned as logical `1` (connected) or `0` (not connected). If the `Connected` status is `0`, that might indicate an issue with the broker; check that you have the correct address, clear the object, and try creating it again.

Example: 1

Data Types: `logical`

Object Functions

subscribe	Subscribe to MQTT topic
unsubscribe	Unsubscribe from MQTT topics
read	Read available messages from MQTT topic
peek	View most recent message from MQTT topic
flush	Clear received MQTT messages
write	Write message to MQTT topic

Examples

Connect to an MQTT Broker

Create a nonsecure MQTT client connection to a HiveMQ public broker with default settings.

```
mqttClient = mqttclient("tcp://broker.hivemq.com")
```

```
mqttClient = 

  Client with properties:

        BrokerAddress: "tcp://broker.hivemq.com"
                 Port: 1883
             ClientID: ""
              Timeout: 5
    KeepAliveDuration: 60
        Subscriptions: [0×3 table]
            Connected: 1
```

Connect with a Specific ID and Port

Create a nonsecure MQTT client connection to a HiveMQ public broker using port 1883 and specify the client ID as `myClient`.

```
mqttClient = mqttclient("tcp://broker.hivemq.com",ClientID="myClient",Port=1883)
```

```
mqttClient = 

  Client with properties:

        BrokerAddress: "tcp://broker.hivemq.com"
                 Port: 1883
             ClientID: "myClient"
              Timeout: 5
    KeepAliveDuration: 60
        Subscriptions: [0×3 table]
            Connected: 1
```

Make a Secure Connection over SSL

Create an MQTT client with a secure connection over SSL using certificates for authentication. Connect the client to the Eclipse Mosquitto™ public broker at port 8884 and specify the broker root certificate, client certificate, and private key.

```
mqttClientSSL = mqttclient("ssl://mosquitto.org",Port=8884,...
        CARootCertificate="C:\mqtt\mosquitto.org.pem",...
        ClientCertificate="C:\mqtt\client.pem",...
        ClientKey="C:\mqtt\client.key")
```

Make a Connection over Websockets

Create an MQTT client connected with websockets to ThingSpeak™. Connecting with the MQTT interface on ThingSpeak requires `ClientID`, `Username`, and `Password`.

```
mqttClient = mqttclient("ws://mqtt3.thingspeak.com",Port=80,...
             Username="MyUserID",ClientID="MyClientID",Password="MyPassword")
```

Version History

Introduced in R2022a

obj2mfile

Convert OPC object to MATLAB code

Syntax

```
obj2mfile(DAObj,'FileName')
obj2mfile(DAObj,'FileName','Syntax')
obj2mfile(DAObj,'FileName','Mode')
obj2mfile(DAObj,'FileName','Syntax','Mode')
```

Description

`obj2mfile(DAObj,'FileName')` converts the `opcda` object `DAObj` to the equivalent MATLAB code using the `set` syntax and saves the MATLAB code to a file specified by `FileName`. If an extension is not specified, the `.m` extension is used. Only those properties that are not set to their default values are written to `FileName`.

`obj2mfile(DAObj,'FileName','`*`Syntax`*`')` converts the OPC object to the equivalent MATLAB code using the specified `'`*`Syntax`*`'` and saves the code to the file, `FileName`. `'`*`Syntax`*`'` can be either `'set'` or `'dot'`. By default, `'set'` is used.

`obj2mfile(DAObj,'FileName','`*`Mode`*`')` and `obj2mfile(DAObj,'FileName','`*`Syntax`*`','`*`Mode`*`')` save the equivalent MATLAB code for all properties if `'`*`Mode`*`'` is `'all'`, and save only the properties that are not set to their default values if `'`*`Mode`*`'` is `'modified'`. By default, `'modified'` is used.

If `DAObj`'s `UserData` is not empty or if any of the callback properties is set to a cell array of values or to a function handle, the data stored in those properties is written to a MAT-file when the toolbox object is converted and saved. The MAT-file has the same name as the file containing the toolbox object code, but with a different extension.

The values of read-only properties will not be restored. For example, if an object is saved with a `Status` property value of `'connected'`, the object will be recreated with a `Status` property value of `'disconnected'` (the default value). You can use `propinfo` to determine if a property is read-only.

To recreate `DAObj`, type the name of the file that you previously created with `obj2mfile`.

Examples

Create a client with a group and an item, then save that client to disk:

```
da = opcda('localhost','Dummy.Server');
da.Tag = 'myopcTag';
da.Timeout = 300;
grp = addgroup(da,'TestGroup');
itm = additem(grp,'Dummy.Tag1');
obj2mfile(da,'myopc.m','dot','all');
```

Recreate the client under a different name:

```
copyOfDA = myopc;
```

Version History

Introduced before R2006a

See Also

`opchelp` | `propinfo`

opccallback

Event information for OPC callbacks

Syntax

```
opccallback(Obj,Event)
```

Description

`opccallback(Obj,Event)` displays a message in the MATLAB Command Window that contains information about an OPC event. The message includes the type of event, the time the event occurred, and the related data for that event.

`Obj` is the object associated with the event. `Event` is a structure that contains the `Type` and `Data` fields. `Type` is the event type. `Data` is a structure containing event-specific information.

`opccallback` is an example callback function. Use this callback function as a template for writing your own callback function. By default, `@opccallback` is the value for the `ReadAsyncFcn`, `WriteAsyncFcn`, and `CancelAsyncFcn` properties of a `dagroup` object, and for the `ErrorFcn` and `ShutDownFcn` properties of an `opcda` object.

Version History

Introduced before R2006a

See Also

`showopcevents`

opcda

Create OPC data access object

Syntax

```
DAobj = opcda(HostID,ServerID)
DAobj = opcda(HostID,ServerID,Name,Value)
```

Description

`DAobj = opcda(HostID,ServerID)` creates an OPC data access object, `DAobj`, for the host specified by `Host` and the OPC server ID specified by `ServerID`. When you create `DAobj`, its initial `Status` property value is `'disconnected'`. To communicate with the server, you must connect `DAobj` to the server with the `connect` function.

`DAobj = opcda(HostID,ServerID,Name,Value)` creates an OPC DA object, `DAobj`, for the host specified by `Host` and the OPC server ID specified by `ServerID`, applying the specified property values. If you specify an invalid property name or value, the function does not create an object.

For a complete listing of OPC functions and properties, type `opchelp`.

Examples

Create OPC DA Clients

These examples show how to create OPC DA clients for local and remote servers.

Create an OPC DA client for a local server.

```
daObj1 = opcda('localhost','Dummy.Server.ID');
```

Create an OPC DA client for a remote server.

```
daObj2 = opcda('ServerHost1','OPCServer.ID');
```

Input Arguments

`HostID` — OPC server host name or IP address
char | string

OPC server host name or IP address, specified as a character vector or string.

Example: `'localhost'`

Data Types: `char` | `string`

`ServerID` — OPC server ID
char | string

OPC server ID, specified as a character vector or string.

Example: `'OPCsrvhost'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The property name-value pairs can be any format that the `set` function supports, i.e., name-value pairs, structures, and name-value cell array pairs. You can specify the writeable properties described in "Output Arguments" on page 19-104, including the following.

Example: `'Timeout',60`

`Timeout` — Maximum time to wait for completion of instruction to server
10 (default)

Maximum time to wait for completion of instruction to server, specified in seconds.

Example: 60

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`UserData` — Data to associate with object
any MATLAB data type

Data to associate with object, specified as any MATLAB data type. `UserData` stores any data that you want to associate with the object. The object does not use this data directly, but you can use the data for identification or other purposes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `string` | `cell`

Output Arguments

`DAobj` — OPC DA client
`opcda` object

OPC DA client, returned as an `opcda` object, with the properties described in opcda Object Properties Properties.

For information on any of these properties, type `opchelp opcda.`*`PropName`*, for example:

```
opchelp opcda.TimerPeriod
```

Version History

Introduced before R2006a

See Also

Functions
`connect` | `opchelp` | `set`

Properties
opcda Object Properties Properties

Topics
“Configure OPC Data Access Object Properties” on page 6-13

OPC Data Access Explorer

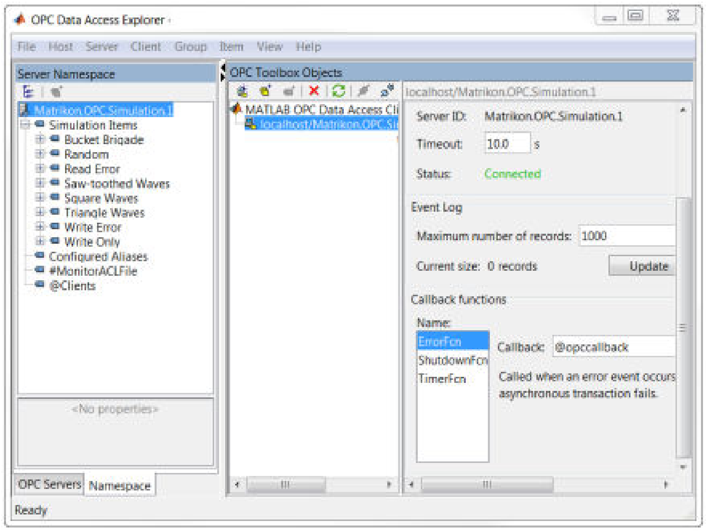
Explore and exchange data with OPC Data Access servers

Description

The **OPC Data Access Explorer** allows you to graphically browse the contents of an OPC server, view server item properties, and create and configure OPC clients, groups, and items in the toolbox.

After browsing, discovery, and configuration, the app allows you to:

- Read and write OPC data.
- Configure and start a logging session, and export logged data to the workspace.
- Export clients, groups, and items to the workspace, to a MAT-file, or as an OPC session file which you can import into the app at a later stage.



Server ID: Matrikon.OPC.Simulation.1
Timeout: 10.0 s
Status: Connected

Event Log
Maximum number of records: 1000
Current size: 0 records

Callback functions
Name: ErrorFcn, ShutdownFcn, TimerFcn
Callback: @opccallback
Called when an error event occurs asynchronous transaction fails.

Open the OPC Data Access Explorer App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app.
- MATLAB command prompt: Enter `opcDataAccessExplorer`.

Examples

- "Access Data with the OPC Data Access Explorer" on page 3-2

Programmatic Use

`opcDataAccessExplorer` opens the **OPC Data Access Explorer** app.

`opcDataAccessExplorer('SessionName')` opens the **OPC Data Access Explorer** app and loads a previously saved OPC session file.

Version History

Introduced before R2006a

Topics

"Access Data with the OPC Data Access Explorer" on page 3-2

opc.daQualityString

OPC data access part of quality ID

Syntax

`[MajorStr,SubStr,LimitStr] = opc.daQualityString(IDs)`

Description

`[MajorStr,SubStr,LimitStr] = opc.daQualityString(IDs)` converts the data access (DA) portion of the OPC quality attribute in `IDs` to the major quality text `MajorStr`, substatus text `SubStr`, and limit text `LimitStr`.

If `IDs` is a vector, each of `MajorStr`, `SubStr`, and `LimitStr` is a cell array the same size as `IDs`.

Examples

Load the OPC HDA example data file and find the qualities of the time stamp union of `hdaDataSmall`:

```
load opcSampleHdaData;
newObj = tsunion(hdaDataSmall);
[majorStr, subStr, limitStr] = opc.daQualityString(newObj.Quality);
```

Version History

Introduced in R2011a

See Also

`opc.hdaQualityString` | `opcqstr`

opc.daSupport

OPC data access troubleshooting utility

Syntax

```
opc.daSupport('localhost')
opc.daSupport('HostName')
opc.daSupport('HostName','FileName')
opc.daSupport('HostName',Fid)
outFile = opc.daSupport(___)
```

Description

`opc.daSupport('localhost')` returns diagnostic information for all OPC data access servers installed on the local machine, and saves the output text to the file `opcsupport.txt` in the current folder. Then the file opens in the editor for viewing.

`opc.daSupport('HostName')` returns diagnostic information for the OPC servers installed on the host with name `HostName`, and saves the output text to the file `opcsupport.txt` in the current folder. Then the file opens in the editor for viewing.

`opc.daSupport('HostName','FileName')` returns diagnostic information for the host with the name `HostName`, and saves the output text to the file `FileName` in the current folder. Then the file opens in the editor for viewing.

`opc.daSupport('HostName',Fid)` appends its output information to the file already opened with `fopen`. The `Fid` argument must be a valid file identifier.

`outFile = opc.daSupport(___)` returns the full path to the generated file and does not open the file in the editor for viewing.

Examples

Get Diagnostics for All Servers on the Local Machine

```
opc.daSupport('localhost')
```

Get Diagnostics for All Servers on Specified Machine

```
opc.daSupport('area1')
```

Save Diagnostic Information to Specified File

```
opc.daSupport('area1','myfile.txt')
```

Input Arguments

`'HostName'` — Machine hosting OPC server
`'localhost'` | other character vector or string

Machine hosting OPC servers, specified as a character vector or string.

Data Types: `char` | `string`

`'FileName'` — File name for output text
`'opcsupport.txt'` (default)

File name for output text, specified as a character vector or string.

Data Types: `char` | `string`

`Fid` — File identifier for open output file
file identifier for the open output file, set by the MATLAB `fopen` function

Example: `Fid = fopen('MyOPCSupport.txt')`

Output Arguments

`outFile` — Path to file of results
character vector

Path to file of results, returned as a character vector.

Version History

Introduced in R2013a

See Also

Functions
`opcserverinfo` | `opc.hdaSupport` | `opcda`

opcfind

Find OPC objects with specific properties

Syntax

```
Out = opcfind
Out = opcfind('P1',V1,'P2',V2,...)
Out = opcfind(S)
```

Description

`Out = opcfind` returns a cell array, `Out`, of all existing OPC objects.

`Out = opcfind('P1',V1,'P2',V2,...)` returns a cell array, `Out`, of toolbox objects whose property values match those passed as property name/property value pairs, *P1*, V1, *P2*, V2, etc.

`Out = opcfind(S)` returns a cell array, `Out`, of toolbox objects whose property values match those defined in structure `S`. The field names of `S` are object property names and the field values of `S` are the requested property values.

Examples

Create some OPC objects:

```
da1 = opcda('localhost','Dummy.ServerA');
da2 = opcda('localhost','Dummy.ServerB');
da1.Tag = 'myopcTag';
da1.Timeout = 300;
grp = addgroup(da2,'TestGroup');
itm = additem(grp,{'Dummy.Tag1','Dummy.Tag2'});
```

Find all OPC objects:

```
allObjCell = opcfind;
```

Find all objects with the Tag `'myopcTag'`:

```
myOPC = opcfind('Tag','myopcTag')
```

Find all `daitem` objects:

```
itmCell = opcfind('Type','daitem')
```

Version History

Introduced in R2006a

See Also

`delete`

opc.getDateDisplayFormat

Format for date display of OPC objects

Syntax

```
fmt = opc.getDateDisplayFormat
```

Description

`fmt = opc.getDateDisplayFormat` returns the current date display format for OPC HDA data objects. The date display format persists across MATLAB sessions.

Examples

Get the current date display format for OPC objects:

```
fmt = opc.getDateDisplayFormat
```

Version History

Introduced in R2011a

See Also

`opc.setDateDisplayFormat`

opchda

Create OPC historical data access client

Syntax

```
hdaObj = opchda(SIObj)
hdaObj = opchda(Hostname,ServerID)
hdaObj = opchda(Hostname,ServerID,Name,Value)
hdaObj = opchda(SIObj,Name,Value)
```

Description

`hdaObj = opchda(SIObj)` constructs an OPC HDA client object, `hdaObj`, for the information provided in the OPC HDA ServerInfo object, `SIObj`, obtained from an `opchdaserverinfo` function call.

`hdaObj = opchda(Hostname,ServerID)` constructs `hdaObj` for the host specified by `Hostname` and the OPC server ID specified by `ServerID`.

When you construct `hdaObj`, its initial `Status` property value is `'disconnected'`. To communicate with the server, connect `hdaObj` to the server using the `connect` function.

`hdaObj = opchda(Hostname,ServerID,Name,Value)` applies the specified property values to the client created with the `Host` and `ServerID` parameters. If you specify an invalid property name or value, the function does not create an object.

`hdaObj = opchda(SIObj,Name,Value)` applies the specified property values to the client created with the ServerInfo object, `SIObj`. If you specify an invalid property name or value, the function does not create an object.

Examples

Create Client Object for a Specific Server

Create an OPC HDA client object for a specific client on the local host.

```
hdaObj = opchda('localhost','MyHDAServer.1');
```

Create Client Objects for All Servers

Create OPC HDA client objects for all clients on the local host.

```
SIObj  = opchdaserverinfo('localhost');
hdaObj = opchda(SIObj);
```

Input Arguments

`SIObj` — OPC HDA server information
OPC HDA ServerInfo object

OPC HDA server information, specified as an OPC HDA ServerInfo object. This object is returned from the function `opchdaserverinfo`.

Example: `SIOjb = opchdaserverinfo`

`Hostname` — OPC HDA server host name
character vector or string

OPC HDA server host name specified as a character vector or string.

Example: `'host-name'`

Data Types: `char` | `string`

`ServerID` — Identifier of OPC HDA server
character vector or string

Identifier of OPC HDA server, specified as a character vector or string.

Example: `'MyHDAServer.1'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

The argument name identifies a property of the created OPC HDA client object. Note that the name-value pairs can be any format that the `set` function supports, i.e., name-value pairs, structures, and name-value cell array pairs.

Example: `'Timeout',60`

`Timeout` — Maximum time to wait for completion of instruction to server
10 (default)

Maximum time to wait for completion of instruction to server, specified in seconds.

Example: 60

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`UserData` — Data to associate with object
any MATLAB data type

Data to associate with object, specified as any MATLAB data type. `UserData` stores any data that you want to associate with the object. The object does not use this data directly, but you can use the data for identification or other purposes.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `string` | `cell`

Output Arguments

`hdaObj` — OPC HDA client
OPC HDA client object

OPC HDA client, returned as an OPC HDA client object.

Version History

Introduced in R2013a

See Also

Functions
`opchdaserverinfo`

opc.hda.Client

Package: `opc.hda`

Create OPC historical data access client

Syntax

```
hdaObj = opc.hda.Client(SIObj)
hdaObj = opc.hda.Client(Host, ServerID)
hdaObj = opc.hda.Client(Host, ServerID, 'P1', V1, 'P2', V2, ...)
hdaObj = opc.hda.Client(SIObj, 'P1', V1, 'P2', V2, ...)
```

Description

`hdaObj = opc.hda.Client(SIObj)` constructs an OPC HDA client object `hdaObj` for the information provided in the OPC HDA ServerInfo object `SIObj` obtained from a `getServerInfo` function call.

`hdaObj = opc.hda.Client(Host, ServerID)` constructs an OPC HDA client object, `hdaObj`, for the host specified by `Host` and the OPC server ID specified by `ServerID`. When you construct `hdaObj`, its initial `Status` property value is `'disconnected'`. To communicate with the server, you must connect `hdaObj` to the server with the `connect` function.

`hdaObj = opc.hda.Client(Host, ServerID, 'P1', V1, 'P2', V2, ...)` applies the specified property values to the client created with the `Host` and `ServerID` parameters. If you specify an invalid property name or value, the function does not create an object.

`hdaObj = opc.hda.Client(SIObj, 'P1', V1, 'P2', V2, ...)` applies the specified property values to the client created with the ServerInfo object `SIObj`. If you specify an invalid property name or value, the function does not create an object. Note that the property name/property value pairs can be any format that the set function supports, i.e., name-value pairs, structures, and name-value cell array pairs.

The OPC HDA client class is responsible for managing connections to an OPC Historical Data Access server. Using the client, you can browse the server's name space, read attributes of items, and read raw or processed data from items on the server.

Examples

Create an HDA client for the Matrikon Simulation Server:

```
hdaObj = opc.hda.Client('localhost', 'Matrikon.OPC.Simulation');
```

Browse the local host for OPC HDA servers and create a client from the first server found:

```
siObj = opc.getServerInfo('localhost');
hdaObj = opc.hda.Client(siObj(1));
```

Version History

Introduced in R2011a

See Also

`connect` | `disconnect` | `opchda` | `opchdaserverinfo`

opc.hda.getServerInfo

Query host for installed HDA servers

Description

`S = opc.hda.getServerInfo('HostName')` queries the host named `HostName` for the OPC HDA servers installed on that host. `'HostName'` can be a host name, or IP address.

`S` is returned as a vector of OPC HDA ServerInfo objects, containing the following read-only properties.

Property Name	Description
`Host`	The host name passed to `getServerInfo`
`ServerID`	The programmatic Server ID to use when constructing an HDA Client object associated with the server
`Description`	A text description of the server
`HDASpecification`	A character vector denoting the HDA specification supported. Currently, only `'HDA1'` will be returned in this property.

Using the `ServerInfo` objects in `S`, you can find a particular server based on the `Description` property using `findDescription(S, 'StartText')`, or you can construct a client by passing the relevant element of `S` to the `opchda` function.

Version History

Introduced in R2011a

See Also

`opchda`

opc.hdaQualityString

OPC historical data access part of quality ID

Syntax

`QStr = opc.hdaQualityString(IDs)`

Description

`QStr = opc.hdaQualityString(IDs)` converts the HDA portion of the OPC quality text in `IDs`.

If `IDs` is a vector, `QStr` is a cell array of character vectors, the same size as `IDs`.

Examples

Load the OPC HDA example data file and find the HDA qualities of the time stamp union of `hdaDataSmall`:

```
load opcSampleHdaData;
newObj = tsunion(hdaDataSmall);
hdaQStr = opc.hdaQualityString([newObj.Quality]);
```

Version History

Introduced in R2011a

See Also

`opc.daQualityString`

opc.hda.reset

Package: `opc.hda`

Disconnect and delete all OPC HDA client objects

Syntax

`opc.hda.reset`

Description

`opc.hda.reset` disconnects and deletes all OPC HDA Client objects. Note that all objects, including those in private work spaces, will be disconnected and deleted when calling this function.

You cannot reconnect an OPC HDA Client object to the server after it has been deleted. Therefore, you should remove it from the workspace with the `clear` function.

Note that `opc.hda.reset` has no influence over OPC Data Access objects. Delete those objects using `opcreset`.

Examples

Create an OPC HDA Client, delete the object using `opc.hda.reset`, and clear the variable from the workspace:

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
opc.hda.reset;
clear hdaObj
```

Version History

Introduced in R2011a

See Also

`clear` | `connect` | `delete`

opchdaserverinfo

Query host for installed HDA servers

Description

`S = opchdaserverinfo('HostName')` queries the host named `HostName` for the OPC HDA servers installed on that host. `'HostName'` can be a host name, or IP address, specified as a character vector or string.

`S` is returned as a vector of OPC HDA ServerInfo objects, containing the following read-only properties.

Property Name	Description
`Host`	The host name passed to `getServerInfo`
`ServerID`	The programmatic Server ID to use when constructing an HDA Client object associated with the server
`Description`	A text description of the server
`HDASpecification`	A character vector denoting the HDA specification supported. Currently, only `'HDA1'` is returned in this property.

Using the `ServerInfo` objects in `S`, you can find a particular server based on the `Description` property using `findDescription(S,'StartText')`, or you can construct a client by passing the relevant element of `S` to the `opchda` function.

Examples

Find a list of HDA servers on the local host.

```
sInfo = opchdaserverinfo('localhost');
```

Locate the specific server with a description containing the character vector `'Matrikon'`.

```
mIndex = findDescription(sInfo,'Matrikon')
```

Construct an OPC HDA client for that server.

```
hdaClient = opchda(sInfo(mIndex))
```

Version History

Introduced in R2014a

See Also

`opchda`

opc.hdaSupport

OPC HDA troubleshooting utility

Syntax

```
opc.hdaSupport('localhost')
opc.hdaSupport('HostName')
opc.hdaSupport('HostName','FileName')
opc.hdaSupport('HostName',Fid)
outFile = opc.hdaSupport( ___ )
```

Description

`opc.hdaSupport('localhost')` returns diagnostic information for all OPC HDA servers installed on the local machine, and saves the output text to the file `opcsupport.txt` in the current folder. Then the file opens in the editor for viewing.

`opc.hdaSupport('HostName')` returns diagnostic information for the OPC HDA servers installed on the host with name `HostName`, and saves the text output to the file, `opcsupport.txt` in the current directory. Then the file opens in the editor for viewing.

`opc.hdaSupport('HostName','FileName')` saves the text output to the file `FileName` in the current folder. Then the file opens in the editor for viewing.

`opc.hdaSupport('HostName',Fid)` appends the output information to the file already opened with `fopen`. The `Fid` argument must be a valid file identifier.

`outFile = opc.hdaSupport( ___ )` returns the full path to the generated file and does not open the file in the editor for viewing. This syntax can use any input arguments previously listed in earlier syntaxes.

Examples

Get Diagnostics for All Servers on the Local Machine

```
opc.hdaSupport('localhost')
```

Get Diagnostics for All Servers on Specified Machine

```
opc.hdaSupport('area1')
```

Save Diagnostic Information to Specified File

```
opc.hdaSupport('area1','myfile.txt')
```

Input Arguments

`'HostName'` — Machine hosting OPC server
`'localhost'` | other character vector or string

Machine hosting OPC servers, specified as a character vector or string.

Data Types: `char` | `string`

`'FileName'` — File for output text
`'opcsupport.txt'` (default)

File for output text, specified as a character vector or string.

Data Types: `char` | `string`

`Fid` — File identifier for open output file
file identifier for open output file, set by the MATLAB `fopen` function

Example: `Fid = fopen('MyOPCSupport.txt')`

Output Arguments

`outFile` — Path to file of results
character vector

Path to file of results, returned as a character vector.

Version History

Introduced in R2013a

See Also

Functions
`opcserverinfo` | `opc.daSupport` | `opchda`

opchelp

Help for OPC data access function or property

Syntax

```
opchelp
opchelp('Name')
Out = opchelp('Name')
opchelp(Obj)
opchelp(Obj,'Name')
Out = opchelp(Obj,'Name')
```

Description

`opchelp` displays a listing of OPC data access functions with a brief description of each function.

`opchelp('Name')` displays online help for the function or property, `Name`. If `Name` is a class, a complete listing of the functions and properties for that class is displayed with a brief description of each. The online help for the object constructor for that class is also displayed. If `Name` is a class with the `.m` extension, then only the online help for the object constructor is displayed.

You can display object-specific function information by specifying `Name` to be `object/function`. For example, to display the online help for the data access object's `connect` function, `Name` would be `'opcda/connect'`.

You can display object-specific property information by specifying `Name` to be `object.property`. For example, to display the online help for the data access object's `Status` property, `Name` would be `'opcda.Status'`.

`Out = opchelp('Name')` returns the help text to the variable `Out`.

`opchelp(Obj)` displays a complete listing of functions and properties for the OPC object `Obj`, along with the online help for the object constructor.

`opchelp(Obj,'Name')` displays the help for function or property, `Name`, for the toolbox object `Obj`.

`Out = opchelp(Obj,'Name')` returns the help text to the variable `Out`.

When displaying property help in the command window, the names in the “See also” section that contain all uppercase letters are function names. The names that contain a mixture of uppercase and lowercase letters are property names.

When displaying function help, the “See also” section contains only function names.

Examples

Display all OPC data access functions and a brief description of each function.

```
opchelp
```

Display help on the `opcda` constructor.

```
daHelp = opchelp('opcda')
```

Display help on the OPC `set` function.

```
opchelp set
```

Display help on the `opcda` object `disconnect` function.

```
opchelp opcda/disconnect
```

Create an `opcda` object and query help information on that object. Get the help for the `Timeout` and `Status` properties.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
opchelp(da)
timeoutHelp = opchelp(da,'Timeout');
opchelp(da,'Status');
```

Version History

Introduced before R2006a

See Also

Functions

`propinfo`

opcqid

Construct quality ID from item quality

Syntax

`QualityID = opcqid(QualityStr)`

Description

`QualityID = opcqid(QualityStr)` returns the quality ID, which is a number between 0 and 255, corresponding to the specified quality attribute. The quality must be a character vector or string in the form `'`*`Major Quality`*`: `*`Quality Sub-status`*` (`*`Limit Status`*`)'`.

If `QualityStr` is an array of quality values, then `QualityID` will be a matrix having the same size as `QualityStr`.

For more information on quality values, see “OPC Quality” on page A-2.

Examples

Construct the quality ID from the quality text of the item `Random.Real8` on the Matrikon OPC Simulation Server:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da)
grp = addgroup(da);
itm = additem(grp,'Random.Real8');
qualityID = opcqid(itm.Quality)
```

Version History

Introduced in R2007b

See Also

Functions
`get` | `opcqstr`

opcqparts

Extract quality parts from OPC quality ID

Syntax

`[MajorQual,Substatus,Limit,Vendor] = opcqparts(QualityID)`

Description

`[MajorQual,Substatus,Limit,Vendor] = opcqparts(QualityID)` extracts the major quality, the quality substatus, the limit status, and the vendor-specific quality information fields, given the `daitem` object QualityID property value.

The QualityID is a double value ranging from `0` to `65535`, made up of four parts. The high 8 bits of the `QualityID` represent the vendor-specific quality information. The low 8 bits are arranged as `QQSSSSLL`, where `QQ` represents the major quality, `SSSS` represents the quality substatus, and `LL` represents the limit status.

For more information on quality values, see “OPC Quality” on page A-2.

Examples

Extract the major quality, substatus, and limit status of the item `Random.Qualities` on the Matrikon OPC Simulation Server:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da)
grp = addgroup(da);
itm = additem(grp,'Random.Qualities');
[quality,substatus,limit] = opcqparts(itm.QualityID)
```

Version History

Introduced before R2006a

See Also

Functions
`get` | `opcqstr`

opcqstr

Convert OPC quality ID into readable text

Syntax

```
QualityStr = opcqstr(QualityID)
```

Description

`QualityStr = opcqstr(QualityID)` constructs a quality character vector from a quality ID, stored in the QualityID property of a `daitem` object. The character vector is of the form `'Major Quality: Quality Substatus: Limit Status'`. The `Limit Status` part is omitted if the limit status is set to `Not Limited`. For information on each of the quality parts, see `opcqparts`.

If QualityID is specified as a vector or matrix of quality IDs, then `QualityStr` will be a cell array having the same size as `QualityID`.

For more information on quality values, see “OPC Quality” on page A-2.

Examples

Construct the quality character vector from the quality ID of the item `Random.Qualities` on a Matrikon OPC Simulation Server:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da)
grp = addgroup(da);
itm = additem(grp,'Random.Qualities');
qualitystr = opcqstr(itm.QualityID)
```

Version History

Introduced before R2006a

See Also

Functions
`get` | `opcqid` | `opcqparts`

opcread

Read logged records from disk to MATLAB workspace

Syntax

```
S = OPCREAD('LogFileName')
S = opcread('LogFileName','PropertyName','PropertyValue',...)
TSCell = opcread('LogFileName','DataType','timeseries')
[I,V,Q,TS,ET] = opcread('LogFileName','DataType',DType,...)
```

Description

`S = OPCREAD('LogFileName')` returns all available records from the OPC log file named `LogFileName`. If no extension is specified as part of `LogFileName`, then `.olf` is used.

`S` is an `NRec`-by-`1` structure array, where `NRec` is the number of records returned. `S` contains the fields `'LocalEventTime'` and `'Items'`. `LocalEventTime` is a date vector corresponding to the local event time for that record. `Items` is an `NItems`-by-`1` structure array containing the fields show below.

Field Name	Description
`ItemID`	The fully qualified item ID, as a character vector.
`Value`	The data value. The data type is dependent on the original Item's `DataType` property.
`Quality`	The data quality, as a character vector.
`TimeStamp`	The time the value was changed, as a date vector.

`S = opcread('LogFileName','`*`PropertyName`*`','PropertyValue',...)` limits the data read from the specified OPC log file based on the properties and values provided. Valid property names and property values are defined in the table below.

Property Name	Property Value
`'Records'`	Specify the required records as `[startRec endRec]`. If no records fall within those bounds, `opcread` returns empty outputs.
`'Dates'`	Specify the date range for records as `[startDt endDt]`. The dates must be in MATLAB date number format. If no records fall within those bounds, `opcread` returns empty outputs.
`'ItemIDs'`	Specify the required item IDs as a character vector, string, or array. If no records match the required `ItemIDs`, `OPCREAD` returns empty outputs.

`TSCell = opcread('LogFileName','DataType','timeseries')` assigns the data received from the OPC log file to a cell array of time series objects. `TSCell` contains as many time series objects as there are items in the group, with the name of each time series object set to the item ID. The quality value stored in the time series object is offset from the quality value returned by the OPC server by 128. The quality displayed by each is the same. Because each record logged might not contain information for every item, the time series objects have only as many data points as there are records containing information about that particular item ID.

`[I,V,Q,TS,ET] = opcread('LogFileName','DataType',`*`DType`*`,...)` assigns the data retrieved from the OPC log file to separate arrays. Valid data types for *`DType`* are `'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'uint32'`, `'logical'`, `'currency'`, `'date'`, and `'cell'`.

`I` is a `1`-by-`NItem` cell array of item names.

`V` is an `NRec`-by-`NItem` array of values with the data type specified. If a data type of `'cell'` is specified, `V` is a cell array containing data in the returned data type for each item. Otherwise, `V` is a numeric array of the specified data type.

Note *`DType`* must be set to `'cell'` when retrieving records containing character vectors or arrays of values.

`Q` is an `NRec`-by-`NItem` array of quality character vectors for each value in `V`.

`TS` is an `NRec`-by-`NItem` array of MATLAB date numbers representing the time when the relevant value and quality were stored on the OPC server.

`ET` is an `NRec`-by-`1` array of MATLAB date numbers, corresponding to the local event time for each record.

Each record logged may not contain information for every item returned, since data for that item may not have changed from the previous update. When data is returned as a numeric matrix, the missing item columns for that record are filled as follows.

`V`	The corresponding value entry is set to the previous value of that item, or to `NaN` if there is no previous value.
`Q`	The corresponding quality entry is set to `'Repeat'`.
`TS`	The corresponding time stamp entry is set to the first valid time stamp for that record.

Examples

Configure and start a logging task. Wait for the task to complete.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExOPCREAD');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-Toothed Waves.Int2');
grp.LoggingMode = 'disk';
grp.RecordsToAcquire = 30;
grp.LogFileName = 'ExOPCREAD.olf';
start(grp);
wait(grp);
```

Retrieve the first two records into a structure:

```
s = opcread('ExOPCREAD.olf','Records',[1, 2]);
```

Retrieve all the data and plot it with a legend:

```
[itmID,val,qual,tStamp] = opcread('ExOPCREAD.olf', ...
    'DataType','double');
```

```
plot(tStamp(:,1),val(:,1),tStamp(:,2),val(:,2));
legend(itmID);
datetick x keeplimits
```

Version History

Introduced before R2006a

See Also

`flushdata` | `getdata` | `peekdata` | `start` | `stop`

opcregister

Install and register OPC Foundation core components

Syntax

```
opcregister
opcregister('repair')
opcregister('remove')
opcregister(..., '-silent')
```

Description

`opcregister` installs the OPC Foundation core components so that the toolbox is able to communicate with OPC servers.

`opcregister('repair')` repairs an existing OPC Foundation core components installation. Use this option if you are experiencing problems querying hosts with the `opcserverinfo` function.

`opcregister('remove')` removes all OPC Foundation core components from your workstation. Use this option if you no longer need to access any servers using OPC.

`opcregister(..., '-silent')` runs the selected option without prompting you for confirmation, and without showing any progress dialog. Note that your machine might be restarted without prompting you if you choose this option. If you are concerned about restarting your machine, do not use the `'-silent'` option.

Note You must clear any OPC objects that you have previously created in this MATLAB session before you can run `opcregister`. If you attempt to run `opcregister` and any OPC objects already exist, an error is generated. Use `opcreset` to clear objects from the MATLAB session.

OPC Foundation core components are redistributed under license from the OPC Foundation, `https://opcfoundation.org`.

Examples

Install the OPC Foundation Core Components

Install the OPC Foundation core components on your local workstation.

```
opcregister
```

Remove the OPC Foundation Core Components

Remove the OPC Foundation core components on your local workstation.

```
opcregister('remove')
```

Version History

Introduced before R2006a

See Also

`opcreset` | `opcserverinfo` | `opchdaserverinfo` | `opcsupport` | `opcfind`

opcreset

Disconnect and delete all OPC objects

Syntax

```
opcreset
opcreset -force
```

Description

`opcreset` disconnects and deletes all OPC objects. This command flushes any data stored in the buffer, cancels all asynchronous operations, and closes any open log files.

You cannot reconnect a toolbox object to the server after you delete the object. Therefore, you should remove these objects from the workspace with the `clear` function.

Note that you cannot call `opcreset` if an OPC Data Access Explorer session is open, or if Simulink models containing OPC blocks are open. Before calling `opcreset`, close all OPC Data Access Explorer sessions and all open Simulink models containing OPC blocks.

`opcreset -force` closes all OPC Data Access Explorer sessions and all Simulink models containing OPC blocks, without prompting to save those sessions and models. If you use the `-force` option, you lose any unsaved changes to those sessions and models. Use the `-force` option only as a last resort.

Examples

Create an `opcda` object, and add a group to that object. Then delete the OPC objects using `opcreset`, and clear all variables from the workspace.

```
da = opcda('localhost','Dummy.Server');
grp = addgroup(da);
opcreset;  % Deletes all objects
% Clear the variables
clear da grp
opcfind
```

Version History

Introduced before R2006a

See Also

`clear` | `delete` | `opcfind`

opcserverinfo

Version, server, and status information

Syntax

```
Out = opcserverinfo
Out = opcserverinfo('Host')
Out = opcserverinfo(DAObj)
```

Description

`Out = opcserverinfo` returns a structure that contains information about installed OPC components, including product version numbers.

`Out = opcserverinfo('Host')` returns a structure that contains OPC server information associated with the host name or IP address specified by `Host`. The information includes the ServerID you can use to create a client associated with that server, and other information about each server.

`Out = opcserverinfo(DAObj)` returns a structure that contains information about the server associated with the `opcda` object `DAObj`. `DAObj` must be a scalar, and must be connected to the server. The information includes the current server status and time information related to the server.

Examples

Retrieve information about servers installed on the local machine:

```
opcserverinfo('localhost')
```

Retrieve information about the Matrikon Simulation Server installed on the local host:

```
da = opcda('localhost', 'Matrikon.OPC.Simulation');
connect(da);
matrikonInfo = opcserverinfo(da)
```

Version History

Introduced before R2006a

See Also

`connect` | `opcda`

opc.setDateDisplayFormat

Set format for date display of OPC objects

Syntax

```
opc.setDateDisplayFormat(DateFmt)
opc.setDateDisplayFormat('default')
NewFmt = opc.setDateDisplayFormat(...)
```

Description

`opc.setDateDisplayFormat(DateFmt)` sets the date display format for OPC HDA data objects to `DateFmt`. `DateFmt` can be any date format number, character vector, or string as defined by `datestr`. The date display format persists across MATLAB sessions.

`opc.setDateDisplayFormat('default')` resets the date display format to the character vector `'yyyy-mm-dd HH:MM:SS.FFF'`.

`NewFmt = opc.setDateDisplayFormat(...)` sets the date display format and returns the new date display format in `NewFmt`.

Examples

Load the OPC HDA example data set and show the values of one of the loaded variables.

```
load opcSampleHdaData;
hdaDataSmall(1).showValues
```

Set the date display format to show time only, and display the values again.

```
opc.setDateDisplayFormat('HH:MM:SS');
hdaDataSmall(1).showValues
```

Reset the display format to the default.

```
dFmt = opc.setDateDisplayFormat('default')
```

Version History

Introduced in R2011a

See Also

Functions
`opc.getDateDisplayFormat`

opcstruct2array

Convert OPC data from structure to array format

Syntax

```
[ItmID,Val,Qual,TStamp,EvtTime] = opcstruct2array(S)
[ItmID,Val,Qual,TStamp,EvtTime] = opcstruct2array(S,'DataType')
```

Description

`[ItmID,Val,Qual,TStamp,EvtTime] = opcstruct2array(S)` converts the OPC data structure `S` into separate arrays for the item ID, value, quality, time stamp, and event time. `S` must be a structure as returned by the `getdata` and `opcread` functions. `S` must contain the fields `LocalEventTime` and `Items`. The `Items` field of `S` must contain the fields `ItemID`, `Value`, `Quality`, and `TimeStamp`.

`ItmID` is a `1`-by-*`nItm`* cell array containing the item IDs of all unique items found in the `ItemID` field of the `Items` structures in `S`.

`Val` is an *`nRec`*-by-*`nItm`* array of doubles containing the value of each item in `ItmID`, at each time specified by `TStamp`.

`Qual` is an *`nRec`*-by-*`nItm`* cell array of character vectors containing the quality of each value in `Val`.

`TStamp` is an *`nRec`*-by-*`nItm`* array of doubles containing the time stamp for each value in `Val`.

`EvtTime` is *`nRec`*-by-`1` array of doubles containing the local time each data change event occurred.

Each row of `Val` represents data from one record received at the corresponding entry in `EvtTime`, while each column of `Val` represents the time series for the corresponding item ID in `ItmID`.

`[ItmID,Val,Qual,TStamp,EvtTime] = opcstruct2array(S,'DataType')` uses the data type specified by the character vector `'DataType'` for the value array. Valid data types are `'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'uint32'`, `'logical'`, `'currency'`, `'date'`, and `'cell'`.

Examples

Configure and start a logging task for 30 seconds of data:

```
da = opcda('localhost', 'Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da, 'ExOPCREAD');
itm1 = additem(grp, 'Triangle Waves.Real8');
itm2 = additem(grp, 'Saw-Toothed Waves.Int2');
grp.LoggingMode = 'memory';
grp.UpdateRate = 0.5;
grp.RecordsToAcquire = 60;
start(grp);
wait(grp);
```

Retrieve the records into a structure:

```
s = getdata(grp);
```

Convert the structure into a `double` array and plot it with a legend:

```
[itmID, val, qual, tStamp] = opcstruct2array(s,'double');
plot(tStamp(:,1), val(:,1), tStamp(:,2), val(:,2));
legend(itmID);
datetick x keeplimits
```

Version History

Introduced before R2006a

See Also

`getdata` | `opcread`

opcstruct2timeseries

Convert OPC data from structure to time series format

Syntax

```
TS = opcstruct2timeseries(S)
```

Description

`TS = opcstruct2timeseries(S)` converts the OPC data structure `S` into a cell array of time series objects. `S` must be a structure in the format that the `getdata` and `opcread` functions return. `S` must contain the fields `LocalEventTime` and `Items`. The `Items` field of `S` must contain the fields `ItemID`, `Value`, `Quality`, and `TimeStamp`.

The cell array `TS` contains as many time series objects as there are unique item IDs in the data structure, with the name of each time series object indicating the item ID. The time series object contains the quality, although this value is offset by 128 from the quality value that the OPC server returns. However, the qualities are the same. Because each logged record might not contain information for every item, the time series objects have only as many data points as there are records containing information about that particular item ID.

Examples

Configure and start a logging task for 30 seconds of data:

```
da = opcda('localhost', 'Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da, 'ExOPCREAD');
itm1 = additem(grp, 'Triangle Waves.Real8');
itm2 = additem(grp, 'Saw-Toothed Waves.Int2');
grp.LoggingMode = 'memory';
grp.UpdateRate = 0.5;
grp.RecordsToAcquire = 60;
start(grp);
wait(grp);
```

Retrieve the records into a structure:

```
s = getdata(grp);
```

Convert the structure into time series objects and plot each separately:

```
ts = opcstruct2timeseries(s);
subplot(2,1,1); plot(ts{1});
subplot(2,1,2); plot(ts{2});
```

Version History

Introduced in R2007b

See Also

Functions
`getdata` | `opcread` | `opcstruct2array`

Topics
"Time Series Objects and Collections"

opcsupport

OPC troubleshooting utility

Syntax

```
opcsupport('localhost')
opcsupport('HostName')
opcsupport('HostName','FileName')
opcsupport('HostName','FileName','da')
opcsupport('HostName','FileName','hda')
outFile = opcsupport(___)
```

Description

`opcsupport('localhost')` returns diagnostic information for all OPC servers installed on the local machine, and saves the output text to the file `opcsupport.txt` in the current folder. This file is then opened in the editor for viewing.

`opcsupport('HostName')` returns diagnostic information for the OPC servers installed on the host with name `HostName`, and saves the text output to the file, `opcsupport.txt` in the current directory. This file is then opened in the editor for viewing.

`opcsupport('HostName','FileName')`, returns diagnostic information for the OPC servers installed on the host with name `HostName`, and saves the text output to the file `FileName` in the current folder. This file is then opened in the editor for viewing.

`opcsupport('HostName','FileName','da')` or `opcsupport('HostName','FileName','hda')` restricts information gathered from the servers on `HostName` to only data access (`'da'`) or to only historical data access (`'hda'`).

`outFile = opcsupport(___)` returns the full path to the generated file and does not open the file in the editor for viewing.

Examples

Get diagnostics for all servers on the local machine

```
opcsupport('localhost')
```

Get diagnostics for all servers on specified machine

```
opcsupport('area1')
```

Save diagnostic information to specified file

```
opcsupport('area1','myfile.txt')
```

Input Arguments

`'HostName'` — Machine hosting OPC server
`'localhost'` | other character vector or string

Machine hosting OPC servers, specified as a character vector or string.

Data Types: `char` | `string`

`'FileName'` — File name for output text
`'opcsupport.txt'` (default)

File name for output text, specified as a character vector or string.

Data Types: `char` | `string`

`'da'` — Indicate data access only
literal character vector or string

Indicate data access only, specified as a literal character vector or string.

Data Types: `char` | `string`

`'hda'` — Indicate historical data access only
literal character vector

Indicate historical data access only, specified as a literal character vector.

Data Types: `char`

Output Arguments

`outFile` — Path to file of results
character vector

Path to file of results, returned as a character vector.

Version History

Introduced before R2006a

See Also

Functions
`opcserverinfo` | `opc.daSupport` | `opc.hdaSupport`

opcua

Create OPC UA client object

Syntax

```
UaClient = opcua(ServerInfoObj)
UaClient = opcua(ServerUrl)
UaClient = opcua(Hostname,Portnum)
```

Description

`UaClient = opcua(ServerInfoObj)` creates an OPC UA client associated with the server specified by `ServerInfoObj`. You can create server objects with the `opcuaserverinfo` function.

`UaClient = opcua(ServerUrl)` creates a client associated with the server referenced by the URL specified in `ServerUrl`.

`UaClient = opcua(Hostname,Portnum)` creates an OPC UA client object associated with the server at port `Portnum` on the host identified by `Hostname`.

By default, the client attempts to retrieve available connection configurations (called Endpoints) from the server and chooses the most secure possible security settings from those configurations. If the attempt to retrieve endpoints fails, an error is generated. You can override the default settings by using `setSecurityModel` to change the `MessageSecurityMode` or `ChannelSecurityPolicy` settings.

Examples

Create OPC UA Clients

Create a client for the first server found on the local host.

```
S = opcuaserverinfo('localhost');
UaClient = opcua(S(1));
```

Create a client for the server at port 53530 on the local host.

```
UaClient = opcua('localhost',53530)
```

```
UaClient = 

OPC UA Client:

   Server Information:
                   Name: 'SimulationServer@localhost'
               Hostname: 'localhost'
                   Port: 53530
            EndpointUrl: 'opc.tcp://localhost:53530/OPCUA/SimulationServer'

   Connection Information:
                Timeout: 10
                 Status: 'Disconnected'
            ServerState: '<Not connected>'

   Security Information:
      MessageSecurityMode: SignAndEncrypt
    ChannelSecurityPolicy: Aes256_Sha256_RsaPss
                Endpoints: [1×11 opc.ua.EndpointDescription]
```

Create a client using the Discovery URL of the server.

```
uaClient = opc.ua.Client('opc.tcp://localhost:53530/OPCUA/SimulationServer');
```

Input Arguments

`ServerInfoObj` — OPC UA server

OPC UA server object

OPC UA server, specified as an OPC UA server object.

Data Types: `object`

`ServerUrl` — OPC UA server URL

char | string

OPC UA server URL, specified as a character vector or string. The server URL must use the `opc.tcp` protocol; Industrial Communication Toolbox does not support http or https connections to an OPC UA server.

Data Types: `char` | `string`

`Hostname` — Server host name or IP address

char | string

Server host name or IP address, specified as a character vector or string. A host name can be short or a fully qualified domain name.

Example: `'localhost'`

Data Types: `char` | `string`

`Portnum` — Server host port number

numeric

Server host port number, specified as a numeric value.

Example: `5000`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

`UaClient` — OPC UA client

`opc.ua.Client` object

OPC UA client, returned as an `opc.ua.Client` object, with the following properties.

Property	Description
`Hostname`	Server host name or IP address
`Port`	Port number used for TCP/IP connections to the server

Property	Description
`Name`	Server description
`Timeout`	Time to wait for all operations on the server to complete
`EndpointUrl`	URL to use for connection to the server
`Namespace`	Server namespace nodes
`UserData`	Free-form container for user-defined data to associate with the client
`MinSampleRate`	Minimum sample rate in seconds that the server can generally support
`AggregateFunctions`	List of aggregate functions supported by this server
`MaxHistoryValuesPerNode`	Maximum history values returned per node in historical read operations
`MaxHistoryReadNodes`	Maximum number of nodes supported by historical read operations
`MaxReadNodes`	Maximum number of nodes supported per read operation
`MaxWriteNodes`	Maximum number of nodes supported per write operation
`MessageSecurityMode`	Message security mode specified for connection
`ChannelSecurityPolicy`	Channel security policy specified for connection
`UserAuthTypes`	User authentication types supported by server

Version History

Introduced in R2015b

See Also

Functions

`opcuaserverinfo` | `connect` | `disconnect` | `setSecurityModel`

opcuanode

Create OPC UA node objects

Syntax

```
NodeList = opcuanode(Index,Id)
NodeList = opcuanode(Index,Id,UaClient)
```

Description

`NodeList = opcuanode(Index,Id)` creates an OPC UA node object or array of objects from the information in `Index` and `Id`. `Index` is a number or numeric vector. `Id` is a character vector, string, scalar integer, or cell array containing character vectors and scalar integers. Use this syntax to create node objects for known nodes on an OPC UA server. Each node `Name` property is set to `'Index:Identifier'`, and other properties of the node are left empty until you use the node to access an OPC UA server. When you successfully use the node object with a client using `writeValue` or `readValue`, the `Client` property of the node is set to the client, and other attributes are read from that client.

`NodeList = opcuanode(Index,Id,UaClient)` immediately associates the node object with the specified client `UaClient`. If `UaClient` is connected at this time, the `opcuanode` function also retrieves other properties from the server associated with `UaClient`.

Use `opcuanode` to create node objects only when you know the index and identifier of nodes you are interested in. For nodes that you need to find from the server, create node objects by browsing the namespace of a connected OPC UA client object with `browseNamespace` or `getNamespace`, or browse the `Parent` and `Children` properties of existing node objects.

Examples

Create a Node and Write to a Server

Construct a node object from index and identifier values. Use the node to write a value to the server, then note that the node has its properties set from the server.

```
S = opcuaserverinfo('localhost');
UaClient = opcua(S);
connect(UaClient);
myNode = opcuanode(2,10225); % Not associated with server yet.
writeValue(UaClient,myNode,pi)
myNode
```

```
myNode = 
OPC UA Node:
   Node Information:
                      Name: 2:10225
               Description: 
            NamespaceIndex: 2
                Identifier: 10225
                  NodeType: Variable
   Hierarchy Information:
                 Parent: ''
               Children: 0

   Server Information:
            ServerDataType: Float
        AccessLevelCurrent: read/write
        AccessLevelHistory: none
               Historizing: 0
```

Create a Node and Browse Other Nodes

Create a known node object and use it to browse for other nodes.

```
UaClient = opcua('localhost',51210);
connect(UaClient);
boilerNode = opcuanode(4,1241,UaClient);
ftxNodes = findNodeByName(boilerNode,'FTX','-partial')
```

```
ftxNodes =
1x2 OPC UA Node array:
    index   Name   NsInd  Identifier  NodeType  Children
    -----  ------  -----  ----------  --------  --------
      1    FTX001  4      1243        Object    1
      2    FTX002  4      1266        Object    1
```

Input Arguments

Index — Node index
numeric value

Node index, specified as a numeric value or array.

Example: 2

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Id — Node ID
numeric | char | string

Node ID, specified as a numeric, character, or string value, or an array of these.

Example: `10225`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`UaClient` — OPC UA client
`opc.ua.Client` object

OPC UA client, specified as an `opc.ua.Client` object. You can create the client using the `opcua` function.

Example: `opcua()`

Output Arguments

`NodeList` — OPC UA nodes
`opc.ua.Node` objects

OPC UA nodes, returned as an array of `opc.ua.Node` objects. An OPC UA node object stores information about a node in an OPC UA server. You can read and write current data, and read historical data using variable nodes. You can browse the name space using object and variable nodes.

A node's type is described by its `NodeType` property, which can indicate an `'Object'` or `'Variable'` type. Variable type nodes can contain data values, while object type nodes cannot contain values. Each node type can contain other nodes: object nodes can contain object and variable nodes, variable nodes can contain other variable nodes.

Node objects include the following properties.

Property	Description
Identity properties	
`Name`	Display name for the node.
`NodeType`	Type of node: `'Object'` or `'Variable'`.
`NamespaceIndex`	Namespace index for this node.
`IdentifierType`	Type of identifier: `'string'`, `'numeric'`, or `'GUID'`.
`Identifier`	Unique identifier. A character vector or integer, depending on the `IdentifierType`.
Relationship properties	
`Parent`	Parent node of this node.
`Children`	Child nodes of this node.
`Client`	Reference to OPC UA client associated with the node.
`FullyQualifiedId`	Character vector that uniquely describes this node.
Essential attributes	
`Description`	Character vector describing the node.
`MinimumSamplingInterval`	Minimum rate at which node value can change.
`Historizing`	True if the server is storing history for the node.
`ServerDataType`	OPC UA data type for node.
Informative attributes	
`AccessLevelCurrent`	User access level to current value: `'none'`, `'read'`, `'write'`, `'read/write'`.
`AccessLevelHistory`	User access level to historical values: `'none'`, `'read'`, `'write'`, `'read/write'`.

Property	Description
`ServerValueRank`	Size restrictions on the server value: `'unrestricted'`, `'scalar'`, `'vector'`, or `'array'`.
`ServerArrayDimensions`	Array dimensions of the server value. Might be empty, as this property is optional for servers.

Version History

Introduced in R2015b

See Also

Functions

`opcua` | `findNodeByName` | `findNodeById` | `browseNamespace` | `readValue` | `getNamespace` | `writeValue`

opcuaserverinfo

Query host for installed OPC UA servers

Syntax

```
Sinfo = opcuaserverinfo(HostName)
Sinfo = opcuaserverinfo(DiscoveryUrl)
```

Description

`Sinfo = opcuaserverinfo(HostName)` queries the specified host for its installed OPC UA servers. `HostName` can be a host name or IP address, specified as a character vector or string.

Note Before running `opcuaserverinfo` to query a host, you must set up a Local Discovery Service (LDS) on that host, as described in Install a Local Discovery Service for OPC UA Server Discovery on page 1-15.

`Sinfo = opcuaserverinfo(DiscoveryUrl)` queries the Discovery Service located at the URL `DiscoveryUrl`. `DiscoveryUrl` must use the `opc.tcp` protocol specified by the syntax `"opc.tcp://hostname:port/Url"`. Use the `DiscoveryUrl` when your server's OPC UA Discovery Service uses a nonstandard port number (by default 4840).

`Sinfo` is returned as an OPC UA ServerInfo object, or an array of these objects, containing the read-only properties `Description`, `Hostname`, `Port`, and `Endpoints`. `Endpoints` contains a list of available endpoints for the server, as an `EndpointDescription` array. `Endpoints` includes information about the security models supported by each endpoint and the user authentication available on that endpoint.

Use the `opcua` function to create an OPC UA client object directly from an `opc.ua.ServerInfo` object.

Examples

Find All Servers

Find all available servers on the local host, and view the properties of the first.

```
Sinfo = opcuaserverinfo('localhost');
Sinfo(1)
```

```
OPC UA ServerInfo 'SimulationServer@tmopti01win1064':

   Connection Information:
               Hostname: 'tmopti01win1064.dhcp.mathworks.com'
                   Port: 53530
              Endpoints: [1×11 opc.ua.EndpointDescription]

   Security Information:
    BestMessageSecurity: SignAndEncrypt
    BestChannelSecurity: Aes256_Sha256_RsaPss
         UserTokenTypes: {'Anonymous'  'Username'  'Certificate'}
```

Create an OPC UA client for the first server found.

```
uaClient = opcua(Sinfo(1));
```

Input Arguments

`HostName` — Host name or IP
char | string

Host name or IP address, specified as a character vector or string, identifying the machine running the OPC UA servers.

Example: `'localhost'`

Data Types: `char` | `string`

`DiscoveryUrl` — Discovery service URL
char | string

Discovery service URL address, specified as a character vector or string in the form `"opc.tcp://hostname:port/Url"`.

Example: `'localhost'`

Data Types: `char` | `string`

Output Arguments

`Sinfo` — Server information
array of `opc.ua.ServerInfo` objects

Server information, returned as an array of `opc.ua.ServerInfo` objects. Index into the array to access the following individual server properties.

Property	Description
`Hostname`	Host name used by server to authenticate connections
`Port`	Port number used for connections to server
`Endpoints`	Available endpoints for server
`BestMessageSecurity`	Highest message security mode supported by server
`BestChannelSecurity`	Most secure channel security policy supported by server
`UserTokenTypes`	List of user authentication types supported by server

Version History

Introduced in R2015b

See Also

Functions
`opcua`

Topics
Install a Local Discovery Service for OPC UA Server Discovery on page 1-15

openosf

Open OPC Data Access Explorer session file

Syntax

```
openosf('Name.osf')
```

Description

`openosf('Name.osf')` opens the OPC Data Access Explorer app and loads the session from the session file `Name.osf`. Specifying the `.osf` extension is optional. `Name.osf` must exist on the MATLAB path, or you must specify the full path to the file.

This function facilitates opening `.osf` files from the file browser window.

Version History

Introduced before R2006a

See Also

Functions
`open`

Apps
OPC Data Access Explorer

peek

Package: `icomm.mqtt`

View most recent message from MQTT topic

Syntax

```
mqttMsg = peek(mqttClient)
mqttMsg = peek(mqttClient,Topic=mqttTopic)
```

Description

`mqttMsg = peek(mqttClient)` returns the most recent message from all subscribed topics for the specified MQTT client, as a timetable of messages. This function does not flush the messages, so you can examine the same messages multiple times.

`mqttMsg = peek(mqttClient,Topic=mqttTopic)` returns the most recent message from the specified topic.

Examples

View Latest MQTT Topic Message

View the most recent message of a subscribed MQTT topic.

```
mqttMsg = peek(mqttClient,Topic="TopMW01")
```

```
mqttMsg =

  1×2 timetable

            Time              Topic            Data
    ____________________    _________    _______________

    14-Dec-2021 16:29:09    "TopMW01"    "Hello World 3"
```

Input Arguments

`mqttClient` — MQTT client
`Client` object

MQTT client specified as an `icomm.mqtt.Client` object, created with the `mqttclient` function.

Example: `mqttClient = mqttclient()`

Data Types: `object`

`mqttTopic` — MQTT topic
string | char

MQTT topic to view the message from, specified as a string or character vector.

Example: `"trubits/mqTop48"`

Data Types: `string` | `char`

Output Arguments

`mqttMsg` — Message viewed from MQTT topic
timetable

Message viewed from MQTT topic, returned as a timetable.

Version History

Introduced in R2022a

See Also

Functions
`mqttclient` | `subscribe` | `unsubscribe` | `read`

peekdata

Preview most recently acquired data

Syntax

```
S = peekdata(GObj,NRec)
```

Description

`S = peekdata(GObj,NRec)` returns the `NRec` most recently acquired records for the `dagroup` object, `GObj`, without removing those records from the toolbox engine. `GObj` must be a scalar `dagroup` object. `S` is a structure array containing data for each record, in the same format as the structure returned by `getdata`.

If `NRec` is greater than the number of records currently available, a warning will be generated and all available records will be returned.

You use `peekdata` when you want to return logged data but you do not want to remove the data from the buffer. The object's `RecordsAvailable` property value will not be affected by the number of samples returned by `peekdata`.

`peekdata` is a non-blocking function that immediately returns records and execution control to the MATLAB workspace.

Examples

Configure and start a logging task for 60 seconds of data.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExOPCREAD');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-Toothed Waves.Int2');
grp.LoggingMode = 'memory';
grp.RecordsToAcquire = 60;
start(grp);
```

Wait for 2 seconds and peek at the two most recent records.

```
pause(2);
s = peekdata(grp,2)
s.Items(1).Value
```

Version History

Introduced before R2006a

See Also

Functions
`flushdata` | `getdata` | `start` | `stop`

piclient

Create OSIsoft PI client

Description

The OSIsoft PI client object provides access to a PI Server so that you can search the server tags and read their data.

Creation

Syntax

```
piClient = piclient(piServer)
piClient = piclient(piServer,Username="WinUserID",Password="WinPwd")
piClient = piclient(
piServer,Username="WinUserID",Password="WinPwd",Domain="WinDomain")
```

Description

`piClient = piclient(piServer)` creates an OSIsoft PI client object `piClient`, and connects to the OSIsoft PI server specified by `piServer`.

`piClient = piclient(piServer,Username="WinUserID",Password="WinPwd")` uses Windows credentials if required by the OSIsoft PI server. `WinUserID` and `WinPwd` are your Windows login name and password, specified as strings. Credential information is used only to connect to the PI server and is not retained in the `piClient` object properties.

`piClient = piclient(`
`piServer,Username="WinUserID",Password="WinPwd",Domain="WinDomain")` specifies the domain name, as a string, associated with the user credentials, if required by the OSIsoft PI server.

If an invalid argument is specified or the connection to the server cannot be established, the object is not created.

Input Arguments

`piServer` — Host name of OSIsoft PI server
string | char

Host name of OSIsoft PI server, specified as a string or character vector.

Example: `"pi-host-55"`

Data Types: `string` | `char`

Properties

`ServerName` — Name of connected OSIsoft PI server
string

This property is read-only.

Name of connected OSIsoft PI server, returned as a string. This is the value provided as the `piServer` input argument to the function.

Example: `"pi-host-55"`

Data Types: `string`

`Domain` — Name of domain associated with user credentials
string

This property is read-only.

Name of domain associated with user credentials, returned as a string. This is the value provided as the `Domain` input argument to the function.

Example: `"MY-NET3"`

Data Types: `string`

Object Functions

tags	List tags from OSIsoft PI server
read	Read data from OSIsoft PI server
viewer	Visualize data from OSIsoft PI Server

Examples

Create a Client Connect to an OSIsoft PI Server

Construct a client object and connect to the OSIsoft PI server named `pi-host-55`.

```
piClient = piclient("pi-host-55");
```

Create a client object and connect to the OSIsoft PI server named `pi-host-55` using Windows credentials.

```
p = piclient("pi-host-55",Username="myID",Password="myPwd");
```

Version History
Introduced in R2022a

plot

Package: `opc.hda`

Plot OPC HDA data object as lines

Syntax

```
plot(dObj)
plot(dObj,'Parent',AX)
plot(dObj, ....)
pH = plot(dObj, ...)
```

Description

`plot(dObj)` plots the data in OPC HDA data object `dObj`. Each element of `dObj` is plotted into the current axes as the value against its time stamp. Quality is not displayed in the plot.

`plot(dObj,'Parent',AX)` plots the data into the axes of handle `AX`.

`plot(dObj, ....)` passes any additional arguments to the MATLAB `plot` function. Use this syntax to define colors and line styles for the data, or to modify other properties of the plotted data.

`pH = plot(dObj, ...)` returns the handles to the created line series objects in `pH`.

In all cases, if the current plot is not held, the X-axis is updated using `datetick` to show date ticks instead of numeric ticks.

Examples

Load the OPC HDA example data file and plot the `hdaDataVis` object:

```
load opcSampleHdaData;
plot(hdaDataVis)
```

See Also

`datetick` | `plot` | `stairs`

propinfo

Property information for OPC objects

Syntax

```
Out = propinfo(Obj)
Out = propinfo(Obj,'PropName')
```

Description

`Out = propinfo(Obj)` returns a structure array with field names given by the property names for `Obj`. Each property name in the output contains a structure with the fields shown below.

Field Name	Description
`Type`	Data type of the property. Possible values are `'any'`, `'callback'`, `'double'`, and `'string'`.
`Constraint`	Type of constraint on the property value. Possible values are `'bounded'`, `'callback'`, `'enum'`, and `'none'`.
`ConstraintValue`	List of valid character vector values or a range of valid values
`DefaultValue`	Default value for the property
`ReadOnly`	Condition under which a property is read-only: • `'always'` — Property cannot be configured. • `'whileConnected'` — Property cannot be configured while `Status` is set to `'connected'`. • `'whileLogging'` — Property cannot be configured while `Logging` is set to `'on'`. • `'never'` — Property can be configured at any time.

`Out = propinfo(Obj,'PropName')` returns a structure array, `Out`, for the property specified by `PropName`. If `PropName` is a cell array of character vectors or an array of strings, then the function returns a cell array of structures for each property.

Examples

```
da = opcda('localhost','Dummy.Server');
allInfo = propinfo(da)
serverIDInfo = propinfo(da,'ServerID')
```

Version History

Introduced before R2006a

See Also

Functions
`opchelp`

read

Read data synchronously from OPC DA groups or items

Syntax

```
S = read(GObj)
S = read(IObj)
S = read(GObj,Src)
S = read(IObj,Src)
```

Description

`S = read(GObj)` and `S = read(IObj)` read data for all the items contained in the `dagroup` object, `GObj`, or for the vector of `daitem` objects, `IObj`. The data is read from the OPC server's cache, and assigned to the structure `S`.

You can synchronously read from the cache only if the `Active` property is set to `'on'` for both the item and the group that contains the item. A warning is issued if any of the objects passed to `read` are inactive. An inactive item is still returned in `S`, but the `Quality` is set to `'BAD: Out of Service'`.

`S = read(GObj,Src)` and `S = read(IObj,Src)` read data from the source specified by `Src`. `Src` can be `'cache'` or `'device'`. If `Src` is `'device'`, data is returned directly from the device. If `Src` is `'cache'`, data is returned from the OPC server's cache, which contains a copy of the device data. Note that the `Active` property is ignored when reading from `'device'`. Note also, that reading data from the device can be slow.

When a `read` operation succeeds, the Value, Quality, and Timestamp properties of the associated items are updated to reflect the values obtained from the read operation.

Examples

Read Data from a Group

This example reads from a device and cache.

Configure a client, group, and item for the Matrikon Simulation Server. Set the update rate for this group to prevent frequent cache updates.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExRead');
grp.UpdateRate = 20;
itm = additem(grp,'Random.Real8');
```

Read twice from the cache, noting that the values are the same each time.

```
v1 = read(grp)
v2 = read(grp)
```

Now read twice from the device, noting that the value updates each time.

```
v3 = read(grp,'device')
v4 = read(grp,'device')
```

Input Arguments

`GObj` — OPC DA group

`dagroup` object

OPC DC group, specified as a `dagroup` object.

Example: `GObj = addgroup()`

`IObj` — OPC DA item

array of `daitem` objects

OPC DA items, specified as an array of opcda item objects.

Example: `IObj = additem()`

`Src` — Data source to read

`'device'` | `'cache'`

Data source to read, specified as `'device'` or `'cache'`.

Example: `'device'`

Data Types: `char` | `string`

Output Arguments

`S` — Read data

structure

Read data, returned as a structure containing data for each item in the following fields:

Field Name	Description	Type
`ItemID`	Fully qualified item name	character vector
`Value`	Value	double, character vector
`Quality`	Quality of the value	character vector
`TimeStamp`	The time that the value and quality was obtained by the device (if this is available), or the time the server updated or validated the value and quality in its cache	Date vector
`Error`	Error message	character vector

Version History

Introduced before R2006a

See Also

Functions
`addgroup` | `additem` | `readasync` | `refresh` | `write` | `writeasync`

read

Package: `icomm.mqtt`

Read available messages from MQTT topic

Syntax

```
read(mqttClient)
mqttMsg = read(mqttClient,Topic=mqttTopic)
```

Description

`mqttMsg = read(mqttClient)` reads all available messages from all subscribed topic in the specified MQTT client. This action flushes the messages so they cannot be read again.

`mqttMsg = read(mqttClient,Topic=mqttTopic)` reads all available messages from the specified MQTT topic among the topics that `mqttClient` is subscribed to.

Examples

Read Messages from MQTT Topics

Read messages from one or more subscribed MQTT topics.

Read all available messages from a specific subscribed topic.

```
mqttMsg = read(mqttClient,Topic="TopMW01");
```

Read all available messages from all subscribed topics.

```
mqttMsg = read(mqttClient)
```

```
mqttMsg =

  2×2 timetable

            Time             Topic          Data
    ____________________    _________    _______________

    14-Dec-2021 16:00:37    "TopMW01"    "Hello World 1"
    14-Dec-2021 16:00:41    "TopMW01"    "Hello World 2"
```

Programmatically access the first message.

```
mqttMsg.Data(1)
```

```
ans = 

    "Hello World 1"
```

Input Arguments

`mqttClient` — MQTT client
`Client` object

MQTT client specified as an `icomm.mqtt.Client` object, created with the `mqttclient` function.

Example: `mqttClient = mqttclient()`

Data Types: `object`

`mqttTopic` — MQTT topic
string | char

MQTT topic to read messages from, specified as a string or character vector.

Example: `"trubits/mqTop48"`

Data Types: `string` | `char`

Output Arguments

`mqttMsg` — Messages read from MQTT topics
timetable

Messages read from MQTT topics, returned as a timetable of messages.

Version History

Introduced in R2022a

See Also

Functions
`mqttclient` | `subscribe` | `unsubscribe` | `peek` | `flush` | `write`

read

Package: `icomm.pi`

Read data from OSIsoft PI server

Syntax

```
piValues = read(piClient,tagName)
piValues = read(piClient,tagName,Earliest=true)
piValues = read(piClient,tagName,DateRange=[startDate,endDate])
piValues = read(piClient,tagName,DateRange=[startDate,endDate],Interval=
timeStep)
```

Description

`piValues = read(piClient,tagName)` reads the latest data value from each of the specified tags of the OSIsoft PI server that the client `piClient` is connected to. `tagName` can be a string or vector of strings. The data is returned to `piValues` as a timetable.

`piValues = read(piClient,tagName,Earliest=true)` reads the earliest value from each of the specified tags.

`piValues = read(piClient,tagName,DateRange=[startDate,endDate])` reads data points in the range between the start and end dates, specified by the datetime values of `[startDate,endDate]`.

`piValues = read(piClient,tagName,DateRange=[startDate,endDate],Interval=timeStep)` reads the data points in the given time range interpolated for each interval specified by the duration `timeStep`.

Examples

Read Various Data Points from OSIsoft PI Server

Read the latest data point of a single tag.

```
piValues = read(piClient, "Plant1_generator1_voltage");
```

Read the earliest data point of a single tag.

```
piValues = read(piClient, "Plant1_generator1_voltage", Earliest=true);
```

Read data points of a single tag over the past one-day period.

```
startDate = datetime("now") - days(1);
endDate = datetime("now");
piValues = read(piClient, "Plant1_generator1_voltage", DateRange=[startDate,endDate]);
```

Read all data points of multiple tags over a 6-hour period, interpolating every 5 minutes.

```
startDate = datetime(2021,6,12,14,10,30); % 12-Jun-2021 14:10:30
endDate = startDate + hours(6);
```

```
piValues = read(piClient,["Plant1_generator1_voltage","Plant1_generator1_current"], ...
          DateRange=[startDate,endDate], Interval=minutes(5));
```

Input Arguments

`piClient` — Client connected to OSIsoft PI server

`icomm.pi.Client` object

Client connected to OSIsoft PI server, specified as an `icomm.pi.Client` object. You create the object with the `piclient` function.

Example: `piClient = piclient(_)`

Data Types: `object`

`tagName` — Tag names to read

string | char

Tag names to read from, specified as a string, string array, character vector, or cell array of character vectors.

Example: `"Power_ckt2"`

Data Types: `char` | `string` | `cell`

`[startDate,endDate]` — Data range of data

vector of datetime

Date range of data to read, specified as a 2-element vector of datetime values.

Example: `[datetime("1-Jan-2020"),datetime("31-Jan-2020")]`

Data Types: `datetime`

`timeStep` — Interval span for data interpolation

duration

Interval span for data interpolation, specified as a duration value.

Example: `hours(1)`

Data Types: `duration`

Output Arguments

`piValues` — Data point values read from PI server

timetable

Data point values read from PI server, returned as a timetable.

Version History

Introduced in R2022a

See Also

Functions
`piclient` | `tags`

read

Read data from Modbus server

Syntax

```
moddata = read(m,target,address)
moddata = read(m,target,address,count)
moddata = read(m,target,address,count,serverId)
moddata = read(m,target,address,count,precision)
moddata = read(m,target,address,count,serverId,precision)
```

Description

`moddata = read(m,target,address)` reads one data value from Modbus object `m` and target area `target` at the starting address `address`. The function reads one value by default.

`moddata = read(m,target,address,count)` reads multiple data values beginning at the starting address `address`. `count` specifies the number of values to read.

`moddata = read(m,target,address,count,serverId)` additionally specifies `serverId`, which is the address of the server to send the read command to.

`moddata = read(m,target,address,count,precision)` additionally specifies the `precision`, which is the data format of the register being read.

`moddata = read(m,target,address,count,serverId,precision)` specifies both the address of the server and the data format precision of the register.

Examples

Read Coils over Modbus

If the read target is coils, the function reads the values from 1–2000 contiguous coils in the remote server, starting at the specified address. A coil is a single output bit. A value of `1` indicates the coil is on and a value of `0` means it is off.

Read 8 coils, starting at address 1. The `address` parameter is the starting address of the coils to read, and the `count` parameter is the number of coils to read.

```
moddata = read(m,'coils',1,8)
```

```
moddata =

   1   1   0   1   1   0   1   0
```

Read Inputs Over Modbus

If the read target is inputs, the function reads the values from 1–2000 contiguous discrete inputs in the remote server, starting at the specified address. A discrete input is a single input bit. A value of `1` indicates the input is on and a value of `0` means it is off.

Read 10 discrete inputs, starting at address 2. The `address` parameter is the starting address of the inputs to read, and the `count` parameter is the number of inputs to read.

```
moddata = read(m,'inputs',2,10)
```

```
moddata =

   1   1   0   1   1   0   1   0   0   1
```

Read Input Registers over Modbus

If the read target is input registers, the function reads the values from 1–125 contiguous input registers in the remote server, starting at the specified address. An input register is a 16-bit read-only register.

Read 4 input registers, starting at address 20. The `address` parameter is the starting address of the input registers to read, and the `count` parameter is the number of input registers to read.

```
moddata = read(m,'inputregs',20,4)
```

```
moddata =

   27640   60013   51918   62881
```

Read Holding Registers over Modbus

If the read target is holding registers, the function reads the values from 1–125 contiguous holding registers in the remote server, starting at the specified address. A holding register is a 16-bit read/write register.

Read 5 holding registers, starting at address 2. The `address` parameter is the starting address of the holding registers to read, and the `count` parameter is the number of holding registers to read.

```
moddata = read(m,'holdingregs',2,5)
```

```
moddata =

   27640   60013   51918   62881   34836
```

Specify Server ID and Precision Options for the Read Operation

You can read any of the four types of targets and also specify the optional parameters for server ID, and you can specify precision for the two types of registers. You can set either option by itself or set both the `serverId` option and the `precision` option together. Both options should be listed after the required arguments.

Read 8 holding registers starting at address 1 using a precision of `'uint32'` from Server ID 3.

```
moddata = read(m,'holdingregs',1,8,3,'uint32');
```

Read Mixed Data Types

You can read contiguous values of different data types (precisions) by specifying the data type for each value. You can do that within the syntax of the `read` function, or set up variables containing arrays of counts and precisions. Both methods are shown here.

Read contiguous registers of the same data type.

```
moddata = read(m,'holdingregs',500,10,'uint32');
```

In that example, the target type is holding registers, the starting address is `500`, the count is `10`, and the precision is `uint32`.

If you wanted to have the 10 values be of mixed data types, you can use this syntax:

```
moddata = read(m,'holdingregs',500,[3 2 3 2],{'uint16', 'single', 'double', 'int16'});
```

Specify both count and precision as arrays of values. In this case, the counts are 3, 2, 3, and 2. The command reads 3 values of data type `uint16`, 2 values of data type `single`, 3 values of data type `double`, and 2 values of data type `int16`. The registers are contiguous, starting at address `500`.

Instead of using arrays inside the read command as shown in the previous step, you can also use arrays as variables in the command. The equivalent code for the same example is:

```
count = [3 2 3 2]
precision = {'uint16', 'single', 'double', 'int16'}
moddata = read(m,'holdingregs',500,count,precision);
```

Using variables is convenient when you have a lot of values to read and they are of mixed data types.

Input Arguments

`target` — Target area to read

character vector | string

Target area to read, specified as a character vector or string. You can perform a Modbus read operation on four types of targets: coils, inputs, input registers, and holding registers, corresponding to the values `'coils'`, `'inputs'`, `'inputregs'`, and `'holdingregs'`. Target must be the first argument after the object name. This example reads 8 coils starting at address 1.

Example: `read(m,'coils',1,8)`

Data Types: `char`

`address` — Starting address to read from

double

Starting address to read from, specified as a double. Address must be the second argument after the object name. This example reads 10 coils starting at address 2.

Example: `read(m,'coils',2,10)`

Data Types: `double`

`count` — Number of values to read
double

Number of values to read, specified as a double. Count is the third argument after the object name. If you do not specify a count, the default of `1` is used. This example reads 12 coils starting at address 2.

Example: `read(m,'coils',2,12)`

Data Types: `double`

`serverId` — Address of the server to send the read command to
double

Address of the server to send the read command to, specified as a double. Server ID must be specified after the object name, target, address, and count. If you do not specify a `serverId`, the default of `1` is used. Valid values are `0-247`, with `0` being the broadcast address.

Note If your device uses a `slaveID` property, it might work to use it as the `serverID` property with the `read` command as described here.

This example reads 8 coils starting at address 1 from server ID 3.

Example: `read(m,'coils',1,8,3);`

Data Types: `double`

`precision` — Data format of the register being read from on the Modbus server
character vector | string

Data format of the register being read from on the Modbus server, specified as a character vector or string. Precision must be specified after the object name, target, address, and count. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional; the default is `'uint16'`.

Note that `precision` does not refer to the return type, which is always `'double'`. It specifies how to interpret the register data.

This example reads 6 holding registers starting at address 2 using a precision of `'uint32'`.

Example: `read(m,'holdingregs',2,6,'uint32');`

Data Types: `char`

Output Arguments

`moddata` — Value of read data
double

Read data values, returned as a double or array of doubles.

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`modbus` | `write` | `writeRead` | `maskWrite`

Topics
“Create a Modbus Connection” on page 18-3
“Configure Properties for Modbus Communication” on page 18-5
“Read Data from a Modbus Server” on page 18-8
“Read Temperature from a Remote Temperature Sensor” on page 18-13

readasync

Read data asynchronously from group or items

Syntax

```
TransID = readasync(GObj)
TransID = readasync(IObj)
```

Description

`TransID = readasync(GObj)` and `TransID = readasync(IObj)` asynchronously read data for all the items contained in the `dagroup` object, `GObj`, or for the vector of `daitem` objects specified by `IObj`. `TransID` is a unique transaction ID for the asynchronous request.

For asynchronous reads, data is always read from the device, not from the server cache. The `Active` property is ignored for asynchronous reads.

When the read operation completes, a read async event is generated by the server. If a callback function file is specified for the `ReadAsyncFcn` property, that function executes when the event is generated.

You can cancel an in-progress asynchronous request using `cancelasync`.

When a `readasync` operation succeeds, the Value, Quality, and Timestamp properties of the associated items are updated to reflect the values obtained from the read operation.

Examples

Configure a client, group, and item, for the Matrikon Simulation Server:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExReadAsync');
grp.UpdateRate = 20;
itm = additem(grp,'Random.Real8');
```

Perform two asynchronous read operations:

```
tid1 = readasync(grp)
tid2 = readasync(grp)
```

Examine the event log:

```
pause(2)
disp('Event log:')
showopcevents(da)
```

Version History

Introduced before R2006a

See Also

`cancelasync` | `read` | `refresh` | `write` | `writeasync`

readAtTime

Package: `opc.hda`

Read data from an OPC HDA server at specified times

Syntax

```
DObj = readAtTime(HdaClient,ItmList,TimeStamps)
[ItmList,Value,Quality,TimeStamp] =
readAtTime(HdaClient,ItmList,TimeStamps,'DataType')
S = readAtTime(HdaClient,ItmList,TimeStamps,'struct')
```

Description

`DObj = readAtTime(HdaClient,ItmList,TimeStamps)` reads data from the items defined by `ItmList`, from the OPC HDA Server associated with client object `HdaClient`, at the time stamps specified by `TimeStamps`. `HdaClient` must be a scalar connected OPC HDA Client. `ItmList` is a character vector, string, or array defining one or more Fully Qualified ItemIDs in the name space of the OPC Server. `TimeStamps` must be a vector of MATLAB date numbers. `DObj` is returned as an `opc.hda.Data` object array the same size as the number of items specified by `ItmList`. Each element of `DObj` is guaranteed to have the same time stamp as the other elements of `DObj`.

When no value exists for a specified time stamp, the server will interpolate a value from the surrounding values to represent the value at that time stamp, and the `Quality` for that time stamp will include the `Interpolated` bit.

`[ItmList,Value,Quality,TimeStamp] = readAtTime(HdaClient,ItmList,TimeStamps,'`*`DataType`*`')` where `'DataType'` is one of the built-in MATLAB numeric arrays (`'double'`, `'single'`, etc.) or `'cell'`, returns the data in the specified data type. `ItmID` is returned as a 1-by-N cell array of character vectors. `Value` is an array of M-by-N values. `Quality` is an array of M-by-N quality IDs, and `TimeStamp` is a M-by-1 array of time stamps as MATLAB date numbers.

`S = readAtTime(HdaClient,ItmList,TimeStamps,'struct')` returns a structure containing the fields `ItemID`, `Value`, `Quality` and `TimeStamp`.

Examples

Create an OPC HDA Client and connect the client to the server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
```

Read the values of two items every 10 seconds for the last hour.

```
DObj = readAtTime(hdaObj,{'Random.Real8','Random.Real4'},[now-1/24:10/86400:now]);
```

Version History

Introduced in R2015b

See Also

Functions
`datenum` | `readRaw` | `readProcessed` | `readModified`

readAtTime

Package: opc.ua

Read historical data from nodes of OPC UA server at specific times

Syntax

```
UaData = readAtTime(UaClient,NodeList,TimeVector)
UaData = readAtTime(NodeList,TimeVector)
```

Description

UaData = readAtTime(UaClient,NodeList,TimeVector) reads stored historical data from the nodes given by NodeList, at the specified times in TimeVector. NodeList is an array of OPC UA node objects, which you can create using getNamespace, browseNamespace, or opcuanode. TimeVector is an array of MATLAB datetimes or date numbers.

UaData is returned as a vector of OPC UA data objects. The server interpolates or extrapolates data if it is not stored at the times specified in TimeVector. Data Quality is set appropriately for interpolated data. If readHistory fails to retrieve history for a given node, that node is not included in the returned OPC UA data object, and a warning is issued. If all requested nodes fail, an error is generated.

UaData = readAtTime(NodeList,TimeVector) reads from the nodes identified by NodeList. All nodes must be of the same connected client.

OPC UA servers provide historical data only from nodes of type Variable. If you attempt to read values from an Object node, no data is returned for that node, the status for that node is set to Bad:AttributeNotSupported, and the node is not included in the returned UaData object.

Examples

Retrieve the 10 minute sampled history for the current day from a local server.

```
uaClnt = opcua('localhost',62550);
connect(uaClnt);
nodeId = '1:Quickstarts.HistoricalAccessServer.Data.Dynamic.Double.txt';
nodeList = opcuanode(2,nodeId,uaClnt);
TimeVector = datetime('today'):minutes(10):datetime('now');
dataObj = readAtTime(uaClnt,nodeList,TimeVector);
```

Version History

Introduced in R2015b

See Also

readValue | readHistory | readProcessed | opcuanode

readHistory

Package: `opc.ua`

Read historical data from nodes on OPC UA server

Syntax

```
UaData = readHistory(UaClient,NodeList,StartTime,EndTime)
UaData = readHistory(UaClient,NodeList,StartTime,EndTime,ReturnBounds)
UaData = readHistory(NodeList,StartTime,EndTime)
UaData = readHistory(NodeList,StartTime,EndTime,ReturnBounds)
```

Description

`UaData = readHistory(UaClient,NodeList,StartTime,EndTime)` reads stored historical data from the nodes identified by `NodeList`, on the server associated with the connected client `UaClient`,with a source timestamp between `StartTime` (inclusive) and `EndTime` (exclusive).`NodeList` is a single OPC UA node object or an array of nodes. `StartTime` and `EndTime` can be MATLAB datetime values or date numbers.

`UaData = readHistory(UaClient,NodeList,StartTime,EndTime,ReturnBounds)` allows you to specify that returned data should include bounding values. Bounding values are the values immediately outside the time range requested (the first value just before `StartTime`, or the first value after `EndTime`) when a value does not exist exactly on the specified limit of the time range. Setting `ReturnBounds` to `true` returns bounding values; setting `ReturnBounds` to `false` (the default) returns values strictly within the specified start and end times.

`UaData = readHistory(NodeList,StartTime,EndTime)` and `UaData = readHistory( NodeList,StartTime,EndTime,ReturnBounds)` read from the nodes identified by `NodeList`. All nodes must be of the same connected client.

Examples

Read History from a Node

This example shows how to retrieve the history for the current day from a local server.

```
uaClnt = opcua('localhost',62550);
connect(uaClnt);
nodeId = '1:Quickstarts.HistoricalAccessServer.Data.Dynamic.Double.txt';
nodeList = opcuanode(2,nodeId,uaClnt);
dataObj = readHistory(uaClnt,nodeList,datetime('today'),datetime('now'));
```

Input Arguments

`UaClient` — OPC UA client

OPC UA client object

OPC UA client specified as an OPC UA client object. The client must be connected.

`NodeList` — List of nodes
array of node objects

List of nodes, specified as an array of node objects or a single node. You can create node objects using `getNamespace`, `browseNamespace`, or `opcuanode`. For information on node object functions and properties, type:

```
help opc.ua.Node
```

You can read only from variable type nodes, not object type nodes. If you specify an object node to read, the return value is an empty array, and the quality is set to `Bad:AttributeIdInvalid`.

`StartTime,EndTime` — Source time span
MATLAB datetime

Source time span, specified as MATLAB datetime values or date numbers. The source times fall between `StartTime` (inclusive) and `EndTime` (exclusive).

`ReturnBounds` — Request bounding values
false (default) | true

Request bounding values, specified as true or false.

Output Arguments

`UaData` — historical data
vector of OPC UA Data objects

Historical data, returned as a vector of OPC UA Data objects. If `readHistory` fails to retrieve history for a given node, that node is not returned in the OPC UA Data object and a warning is issued. If all requested nodes fail, an error is generated.

Version History

Introduced in R2015b

See Also

Functions
`readValue` | `readAtTime` | `opcuanode` | `readProcessed`

readItemAttributes

Package: `opc.hda`

Read item attribute values from OPC HDA server

Syntax

`S = readItemAttributes(HdaObj, ItemID, Attribute, StartTime, EndTime)`

Description

`S = readItemAttributes(HdaObj, ItemID, Attribute, StartTime, EndTime)` reads item attribute values for the `opc.hda.ItemAttributes` item with ID `ItemID`. `HdaObj` must be a scalar OPC HDA client that is already connected to the server.

`ItemID` is a character vector or string containing the item ID for which attributes are requested. `Attribute` is the requested attribute for the item, specified either as a character vector or string as the ID for that attribute. `StartTime` and `EndTime` are MATLAB date numbers representing the start and end times of the period over which data must be aggregated.

`S` is returned as a structure array containing fields `ItemID`, `AttributeID`, `TimeStamp` and `Value`. `ItemID` is the item ID requested. `AttributeID` is the numeric ID of the attribute requested. `TimeStamp` is a vector containing the time stamps when the attribute was updated. `Value` is the value that the attribute was changed to at each time in `TimeStamp`.

The `ItemAttributes` property of the connected client object `HdaObj` contains all valid item attributes for the server.

Examples

Retrieve the current data type of the `'Random.Real8'` property:

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
attrStruct = hdaObj.readItemAttributes('Random.Real8', ...
                hdaObj.ItemAttributes.DATA_TYPE,now,now)
```

Version History
Introduced in R2011a

readModified

Package: `opc.hda`

Read modified data from an OPC HDA server

Syntax

```
DObj = readModified(HdaClient,ItmList,StartTime,EndTime)
```

Description

`DObj = readModified(HdaClient,ItmList,StartTime,EndTime)` reads modified data from the items defined by `ItmList`, stored on the OPC HDA server connected to OPC HDA Client `HdaClient`, between `StartTime` (inclusive) and `EndTime` (exclusive). The `StartTime` and `EndTime` arguments must be date numbers, or character vectors that can be converted to a MATLAB date number. `DObj` is returned as an `opc.hda.Data` array, with one element per item specified in `ItmList`.

`DObj` contains only data items that have been modified, replaced, or deleted on the OPC HDA server; that is, only data values that return a quality of `'Extra Data'` during a `readRaw` operation. If a value has been modified multiple times, all values for that time are returned.

Some servers do not support this function.

Version History

Introduced in R2011a

See Also

Functions
`datenum` | `readRaw`

readProcessed

Package: `opc.hda`

Read server-aggregated data from an OPC HDA server

Syntax

```
DObj =
readProcessed(HdaObj,ItmList,AggregateType,AggregateInterval,StartTime,EndTim
e)
[ItmID,Value,Quality,TimeStamp] =
readProcessed(HdaObj,ItmList,AggregateType,AggregateInterval,StartTime,EndTim
e,'DataType')
S =
readProcessed(HdaObj,ItmList,AggregateType,AggregateInterval,StartTime,EndTim
e,'struct')
```

Description

`DObj = readProcessed(HdaObj,ItmList,AggregateType,AggregateInterval,StartTime,EndTime)` reads processed data from the OPC HDA Server associated with client object `HdaObj`, returning the processed data in `opc.hda.Data` object `DObj`. `HdaObj` must be a scalar OPC HDA client that is already connected to the server.

`ItmList` is a string array or cell array of item IDs to read from. `AggregateType` is the requested aggregate type, obtained from the client's `Aggregates` property. `AggregateInterval` is the time interval in seconds that the server must aggregate data over. `StartTime` and `EndTime` are MATLAB date numbers representing the start and end times of the period over which data must be aggregated.

`[ItmID,Value,Quality,TimeStamp] = readProcessed(HdaObj,ItmList,AggregateType,AggregateInterval,StartTime,EndTime,'`*`DataType`*`')` returns the processed data as separate arrays. `'DataType'` is one of the built-in MATLAB numeric arrays (`'double'`, `'single'`, etc.) or `'cell'`. `ItmID` is returned as a 1-by-N cell array of character vectors. `Value` is an array of M-by-N values. `Quality` is an array of M-by-N quality IDs, and `TimeStamp` is a M-by-1 array of time stamps as MATLAB date numbers.

`S = readProcessed(HdaObj,ItmList,AggregateType,AggregateInterval,StartTime,EndTime,'struct')` returns the processed data as a structure containing fields `ItemID`, `Value`, `Quality` and `TimeStamp`.

Examples

Create an OPC HDA Client and connect the client to the server:

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
```

Read the one minute average values of two items for the last hour:

```
aggregates = hdaObj.Aggregates
DObj = readProcessed(hdaObj,{'Random.Real8','Random.Real4'}, ...
         aggregates.TIMEAVERAGE,60,now-1/24,now);
```

Version History

Introduced in R2011a

See Also

Functions

`readRaw` | `readAtTime` | `readModified` | `opc.hdaQualityString`

readProcessed

Package: `opc.ua`

Read aggregate data from nodes of an OPC UA server

Syntax

```
UaData = readProcessed(UaClient,NodeList,AggregateFn,AggrInterval,
StartTime,EndTime)
UaData = readProcessed(NodeList,AggregateFn,AggrInterval,StartTime,EndTime)
```

Description

`UaData = readProcessed(UaClient,NodeList,AggregateFn,AggrInterval,StartTime,EndTime)` reads processed historical data from the nodes given by `NodeList`. `NodeList` must be an array of OPC UA node objects, which you can create using `getNamespace`, `browseNamespace`, or `opcuanode`. The interval between `StartTime` and `EndTime` (which can be datetime variables or date numbers) is split into intervals of `AggrInterval`, a MATLAB duration variable or a double representing the interval in seconds. For each interval of time, the server calculates a processed value based on the `AggregateFn` requested. `AggregateFn` can be specified as a character vector or as an `AggregateFnId` object. A client stores the available Aggregates for a server in the `AggregateFunctions` property. For a description of Aggregate functions, see “OPC UA Aggregate Functions” on page 17-10.

`UaData` is returned as a vector of OPC UA data objects. If `readProcessed` fails to retrieve historical data for a given node, that node is not included in the returned OPC UA data object, and a warning is issued. If all requested nodes fail, an error is generated.

`UaData = readProcessed(NodeList,AggregateFn,AggrInterval,StartTime,EndTime)` reads from the nodes identified by `NodeList`. All nodes must be of the same connected client.

OPC UA servers provide historical data only from nodes of type `Variable`. If you attempt to read values from an `Object` node, no data is returned for that node, and the status for that node is set to `Bad:AttributeNotSupported`, a warning is issued, and the node is not included in the returned `UaData` object.

Examples

Read Processed Data at Fixed Intervals

Retrieve the average value for each 10 minute interval of the current day from a local server.

```
uaClnt = opcua('localhost',62550);
connect(uaClnt);
nodeId = '1:Quickstarts.HistoricalAccessServer.Data.Dynamic.Double.txt';
```

```
nodeList = opcuanode(2,nodeId,uaClnt);
dataObj = readProcessed(uaClnt,nodeList,'Average',minutes(10),datetime('today'),datetime('now'));
```

Input Arguments

UaClient — OPC UA client
opc.ua.Client object

OPC UA client, specified as an `opc.ua.Client` object. Create a client object with the `opcua` function. The client must be connected.

Example: `opcua()`

NodeList — OPC UA nodes
opc.ua.Node object

OPC UA nodes, specified as a `opc.ua.Node` object, or array of objects.

Example: `opcuanode()`

AggregateFn — Aggregate function
char vector | AggregateFnId object

Aggregate function, specified as a character vector or as an `AggregateFnId` object. A client stores the available aggregates for a server in its `AggregateFunctions` property.

For a description of the standard aggregate functions defined by the OPC Foundation, see “OPC UA Aggregate Functions” on page 17-10.

Example: `'Average'`

AggrInterval — Aggregation interval segment
double | duration

Aggregation interval segment, specified as a MATLAB duration or a double indicating seconds.

Example: `minutes(10)`

Data Types: `double` | `duration`

StartTime,EndTime — Aggregation interval boundaries
datetime | date

Aggregation interval boundaries, specified as datetime or date numbers.

Example: `datetime('today'),datetime('now')`

Data Types: `double` | `datetime`

Output Arguments

UaData — OPC UA data
vector of opc.ua.Data objects

OPC UA data, returned as a vector of `opc.ua.Data` objects. If `readProcessed` fails to retrieve data for a given node, that node is not returned in the `opc.ua.Data` object and a warning is issued. If all requested nodes fail, an error is generated.

Version History

Introduced in R2015b

See Also

Functions

`opcuanode` | `readAtTime` | `readHistory` | `readValue`

readRaw

Package: `opc.hda`

Read raw data of specified time range from HDA server

Syntax

```
DObj = readRaw(HdaClient,ItmList,StartTime,EndTime)
DObj = readRaw(HdaClient,ItmList,StartTime,EndTime,ExtendedBounds)
```

Description

`DObj = readRaw(HdaClient,ItmList,StartTime,EndTime)` reads data from the items defined by `ItmList`, stored on the OPC HDA server connected to OPC HDA client `HdaClient`, between `StartTime` (inclusive) and `EndTime` (exclusive). The `StartTime` and `EndTime` arguments must be date numbers, or character vectors that can be converted to a MATLAB date number. `DObj` is returned as an `opc.hda.Data` array, with one element per item specified in `ItmList`.

`DObj = readRaw(HdaClient,ItmList,StartTime,EndTime,ExtendedBounds)` allows you to specify boundary extension. If `ExtendedBounds` is `true`, then the first data point on or outside the defined start and end times is returned. If `ExtendedBounds` is `false`, then only values that were timestamped between `StartTime` (inclusive) and `EndTime` (exclusive) are included.

One or more timestamps returned for each item can be unique to that item. To retrieve aligned data from an OPC HDA server, use `readAtTime` or `readProcessed`.

Examples

Read Past Day's Data from Two Items

Read data over the past day from two items in the OPC HDA server.

Create an OPC HDA client and connect the client to the server.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation');
connect(hdaObj);
```

Read the last day’s data from two specified items.

```
DObj = readRaw(hdaObj,{'Random.Real8','Random.Real4'},now-1,now);
```

Input Arguments

`HdaClient` — OPC HDA client
OPC HDA client object

OPC HDA client, specified as an OPC HDA client object.

Example: `opchda()`

`ItmList` — HDA items

char vector | string | cell

HDA items, specified as a character vector, string, or supporting array of either.

Example: `{'Random.Real8','Random.Real4'}`

Data Types: `char` | `string` | `cell`

`StartTime,EndTime` — Time boundaries

serial date numbers | character vectors

Time boundaries, specified as serial date numbers. The values can be doubles, such as values returned by the `datenum` or `now` functions, or character vectors that can be converted to date numbers.

Example: `datenum(2018,11,30)`

Data Types: `double` | `char`

`ExtendedBounds` — Extend time boundaries to assure inclusion of start and stop times

false (default) | true

Extend time boundaries to assure inclusion of start and end times, specified as `false` or `true`.

This flag instructs the historical server whether to completely span the required start and end time, or return only values that are contained within the specified start and end time (start time included but not end time). If `ExtendedBounds` is true, the returned values are guaranteed to include the timestamp at or before the specified start time, and at or after the specified end time. If `ExtendedBounds` is false, there is no guarantee that the values include the exact specified start time, and definitely do not include the specified end time. The rules applied by this flag are:

- If `ExtendedBounds` is `false` (default), the server returns all recorded data from the start time up to, but not including, the end time.
- If `ExtendedBounds` is `true`, the server returns all data from the start time up to (and including) the end time. If no data value exists exactly at the start time, the previous value is returned; if no data value exists exactly at the end time, the next value is be returned; even if these values are outside the specified start and end times.
- If `ExtendedBounds` is `true` and no data exists on the start time or before it, the server includes a value of `Empty` at the start timestamp, and a quality of `OPCHDA_NOBOUND` (“No Bound”). Similarly for the end time.

Example: `true`

Data Types: `logical`

Output Arguments

`DObj` — Raw OPC HDA data

OPC HDA data object

Raw OPC HDA data returned as an array of OPC HDA data objects, with one element per item.

Version History

Introduced in R2011a

See Also

Functions

`datenum` | `readAtTime` | `readProcessed` | `readModified`

readValue

Package: `opc.ua`

Read values from nodes on OPC UA server

Syntax

```
[Values,Timestamps,Qualities] = readValue(UaClient,NodeList)
[Values,Timestamps,Qualities] = readValue(NodeList)
```

Description

`[Values,Timestamps,Qualities] = readValue(UaClient,NodeList)` reads the value, quality, and timestamp from the nodes identified by `NodeList`, on the server associated with the connected client `UaClient`.`NodeList` can be a single OPC UA node object or an array of nodes.

`[Values,Timestamps,Qualities] = readValue(NodeList)` reads from the nodes identified by `NodeList`. All nodes must be of the same connected client.

Examples

Read Value from Nodes

Read the current value from a node identified by its `Index` and `Identifier`.

```
UaClient = opcua('localhost',53530);
connect(UaClient);
sineNode = opcuanode(3,'Sinusoid',UaClient);
[val,ts,qual] = readValue(UaClient,sineNode)
```

Read from multiple nodes.

```
simNode = findNodeByName(UaClient.Namespace,'Simulation');
simChildNodes = simNode.Children;
[val,ts,qual] = readValue(UaClient,simChildNodes)
```

Input Arguments

`UaClient` — OPC UA client
OPC UA client object

OPC UA client, specified as an OPC UA client object. The client must be connected.

`NodeList` — List of nodes
array of node objects

List of nodes, specified as an array of node objects or a single node. You can create node objects using `getNamespace`, `browseNamespace`, or `opcuanode`. For information on node object functions and properties, type:

```
help opc.ua.Node
```

You can read only from variable type nodes, not object type nodes. If you specify an object node to read, the return value is an empty array, and the quality is set to `Bad:AttributeIdInvalid`.

Output Arguments

`Values` — Node values
node data type

Node values, returned as node data type, or a cell array of node data types if `NodeList` is an array.. For information about how MATLAB interprets these formats, type:

```
help opc.ua.DataTypeId
```

`Timestamps` — Time of node data source
vector of MATLAB datetime

Time of node data source, returned as a vector of MATLAB datetime objects. Timestamps represent the time that the source provided the data to the server.

`Qualities` — Node data quality
array of OPC UA qualities

Node data quality, returned as an array of OPC UA qualities. For information on OPC UA qualities, type:

```
 help opc.ua.QualityId
```

Version History

Introduced in R2015b

See Also

Functions
`getNamespace` | `browseNamespace` | `opcuanode` | `writeValue`

refresh

Read all active items in group

Syntax

```
refresh(GObj)
refresh(GObj,'Source')
```

Description

`refresh(GObj)` asynchronously reads data for all active items contained in the `dagroup` object specified by `GObj`. Items whose `Active` property is set to `'off'` will not be read. `GObj` can be an array of group objects. The data is read from the OPC server's cache. You can use `refresh` only if the `Active` property is set to `'on'` for `GObj`.

When the refresh operation completes, a `DataChange` event is generated by the server. If a callback function file is specified for the `DataChangeFcn` property, then the function executes when the event is generated.

`refresh` is a special case of subscription that forces a `DataChange` event for all active items even if the data has not changed. Note that `refresh` ignores the `Subscription` property.

`refresh(GObj,'Source')` asynchronously reads data from the source specified by `'Source'`, which can be `'cache'` or `'device'`. If `'Source'` is `'device'`, data is returned directly from the device. If `'Source'` is `'cache'`, data is returned from the OPC server’s cache. Note that reading data from the device can be slow.

Examples

Configure a client, group, and item, for the Matrikon Simulation Server.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExRefresh');
itm = additem(grp,'Random.Real8');
```

Turn off subscription for the group and add a `DataChangeFcn` callback.

```
grp.Subscription = 'off';
grp.DataChangeFcn = 'disp(grp.Item)'
```

Call `refresh` to get group and item updates.

```
refresh(grp)
refresh(grp)
```

Version History

Introduced before R2006a

See Also

Functions
`read` | `readasync` | `write` | `writeasync`

removepublicgroup

Remove public group from server

Syntax

`removepublicgroup(DAObj,'PublicGroupName')`

Description

`removepublicgroup(DAObj,'PublicGroupName')` removes the public group `PublicGroupName` from the server that `DAObj` is connected to. `DAObj` must be a connected `opcda` object.

If the public group has clients using that group, `removepublicgroup` issues a warning; then it removes the group from the server only when all clients have stopped using that group. No additional clients can connect to that group after you call `removepublicgroup`.

Not all OPC data access servers support public groups. If you try to make a public group on a server that does not support public groups, you get an error. To verify that a server supports public groups, use the `opcserverinfo` function on the client connected to that server: Look for an entry `'IOPCPublicGroups'` in the `'SupportedInterfaces'` field.

Examples

Remove Public Group from Server

Connect to the server `Dummy.Server` and remove the public group named `PGroup`.

```
da = opcda('localhost','Dummy.Server');
connect(da);
removepublicgroup(da,'PGroup');
```

Version History

Introduced before R2006a

See Also

Functions
`addgroup` | `makepublic`

resample

Package: `opc.hda`

Resample OPC HDA data object to have defined time stamps

Syntax

```
NewObj = resample(DObj,NewTS)
NewObj = resample(DObj,NewTS,'linear')
NewObj = resample(DObj,NewTS,'hold')
NewObj = resample(DObj,NewTS,'nearest')
NewObj = resample(DObj,NewTS,'spline')
NewObj = resample(DObj,NewTS,'pchip')
```

Description

`NewObj = resample(DObj,NewTS)` resamples data in OPC HDA data object `DObj` so that all elements of the object have the time stamps given by `NewTS`. `NewTS` must be a vector of MATLAB date numbers.

If `DObj` contains elements with the same item ID, those elements are combined into one element. So the size of `NewObj` might be smaller than the size of `DObj`.

Values are linearly interpolated or extrapolated to the new time stamps.

Quality for the resampled data is set as follows:

- All original values retain their quality.
- All interpolated values get a quality of `Interpolated: Good`.
- All extrapolated values get a quality of `Interpolated: Sub-Normal`.

`NewObj = resample(DObj,NewTS,'linear')` uses linear interpolation.

`NewObj = resample(DObj,NewTS,'hold')` uses a zero-order hold interpolation where the previous known value is used for all new time stamps. Any time stamp prior to the first known value is set to `NaN` (or `0` if the value is a fixed-point data type).

`NewObj = resample(DObj,NewTS,'nearest')` uses nearest-neighbor interpolation as defined by `interp1`.

`NewObj = resample(DObj,NewTS,'spline')` uses spline interpolation as defined by `interp1`.

`NewObj = resample(DObj,NewTS,'pchip')` uses shape-preserving, piece-wise, cubic interpolation as defined by `interp1`.

Examples

Load the OPC HDA example data file and resample the first element of `hdaDataSmall`.

```
load opcSampleHdaData;
newTS = datenum(2010,6,1,9,30,0:10:60);
newObj = resample(hdaDataSmall(1),newTS);
```

Display the values and qualities of the new object.

```
newObj.showValues
```

See Also

Functions
`interp1` | `showValues` | `tsintersect` | `tsunion`

save

Save OPC objects to MAT-file

Syntax

```
save FileName
save FileName Obj1 Obj2 ...
save('FileName','Obj1','Obj2',___)
```

Description

`save FileName` saves all variables in the MATLAB workspace to the specified MAT-file, `FileName`. If an extension is not specified for `FileName`, then a `.mat` extension is used.

`save FileName Obj1 Obj2 ...` saves OPC objects, `Obj1`, `Obj2`, `...` to the specified MAT-file, `FileName`. If an extension is not specified for `FileName`, then a `.mat` extension is used.

`save('FileName','Obj1','Obj2',___)` provides the functional form of syntax. When using the functional form, you must specify the file name and toolbox objects as character vectors or strings.

Any data associated with the toolbox object will not be stored in the MAT-file. The data can be brought into the MATLAB workspace with `getdata` and then saved to the MAT-file using a separate variable name.

The `load` command is used to return variables from the MAT-file to the MATLAB workspace. Values for read-only properties will be restored to their default values upon loading. For example, the Status property for an `opcda` object will be restored to `'disconnected'`. You use `propinfo` to determine if a property is read-only.

Examples

Create a connected client and configure a group with two items. Then save the group.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ClearEventLogEx');
itm1 = additem(grp,'Random.Real8');
save mygroup grp
```

Version History

Introduced before R2006a

See Also

Functions
`getdata` | `load` | `opchelp` | `propinfo`

serveritemprops

Property information for items in OPC server name space

Syntax

```
S = serveritemprops(DAObj,ItemID)
S = serveritemprops(DAObj,ItemID,PropID)
```

Description

`S = serveritemprops(DAObj,ItemID)` returns all property information for the OPC server items specified by `ItemID`. `ItemID` is a single, fully qualified ItemID, specified as a character vector or string. `DAObj` is an `opcda` object connected to the OPC server. `S` is a structure array with the following fields:

Field Name	Description
`PropID`	The property number
`PropDescription`	The property description
`PropValue`	The property value

The number of properties returned by the server may be different for different ItemIDs.

Item properties include the item's canonical data type, limits, description, current value, etc.

`S = serveritemprops(DAObj,ItemID,PropID)` returns property information for the property IDs contained in `PropID`. `PropID` is a vector of integers. If `PropID` contains IDs that do not exist for that property, a warning is issued and any remaining property information is returned.

Note This function is not intended to read large amounts of data. Instead, it allows you to easily browse and read small amounts of data specific to a particular ItemID.

For a complete list of Property IDs defined by the OPC Foundation, consult “OPC DA Server Item Properties” on page B-2.

Examples

Find the properties of the Matrikon Simulation Server `Random.Real4` tag.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
p = serveritemprops(da,'Random.Real4');
```

Read the first property to see the item canonical data type.

```
p(1)
```

Read the third property to see the item quality.

```
p(3)
```

Version History

Introduced before R2006a

See Also

Functions

`serveritems`

serveritems

Query server or name space for fully qualified item IDs

Syntax

```
FQID = serveritems(DAObj,ItemID)
FQID = serveritems(DAObj)
FQID = serveritems(DAObj, 'Filter1',Val1,'Filter2',Val2, ...)
FQID = serveritems(NS)
FQID = serveritems(NS,ItemID)
```

Description

`FQID = serveritems(DAObj,ItemID)` returns a cell array of all fully qualified item IDs matching `ItemID` that are found on the OPC server defined by `DAObj`. `DAObj` must be a connected `opcda` object. `ItemID` is a partial character vector or string to search for, and can contain the wildcard character `'*'`. `FQID` is a character vector or cell array of character vectors. You can use `FQID` in a call to `additem` to construct `daitem` objects.

`FQID = serveritems(DAObj)` returns all fully qualified item IDs on the OPC server associated with `DAObj`.

`FQID = serveritems(DAObj, 'Filter1',Val1,'Filter2',Val2, ...)` allows you to filter the retrieved name space based on a number of available browse filters. Available filters are described in the following table:

Browse Filter	Description
`'StartItemID'`	Specify the `FullyQualifiedID` of a branch node, as a character vector or string. Only nodes contained in that branch node will be returned. Some OPC servers do not support partial name space retrieval based on this option: An error is generated if you attempt to use the `'StartItemID'` browse filter on such a server.
`'Depth'`	Specify the depth of the name space that you want returned. A `'Depth'` value of `1` returns only the nodes contained in the starting position. A `'Depth'` value of `2` returns the nodes contained in the starting position and all of their nodes. A `'Depth'` value of `Inf` returns all nodes.
`'AccessRights'`	Restricts the search to leaf nodes with particular access right characteristics. Specify `'read'` to return nodes that include the read access right, and `'write'` to return nodes that include the write access right. An empty character vector (`''`) returns nodes with any access rights.
`'DataType'`	Restricts the search to nodes with a particular canonical data type. Valid data types are `'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'uint32'`, `'logical'`, `'currency'`, and `'date'`. Use the `'DataType'` filter to find server items with a specific data type, such as `'double'` or `'date'`.

`FQID = serveritems(NS)` and `FQID = serveritems(NS,ItemID)` search the name space structure defined by `NS`, rather than querying the OPC server. `NS` is the result of a call to `getnamespace` in either hierarchical or flat format.

Note that some servers may return item IDs that cannot be created on that server. These item IDs are usually branches of the OPC server name space.

You use the results of a call to `serveritems` in a call to `serveritemprops` to return the property information for items in the OPC server name space. The properties of the items in the server name space include the server item's canonical data type, limits, description, current value, etc.

Examples

Create a client for the Matrikon Simulation Server and connect to the server:

```
da = opcda('localhost', 'Matrikon.OPC.Simulation');connect(da);
```

Find all item IDs in the Matrikon Server containing the word `'Real'`:

```
realItmIDs = serveritems(da, '*Real*'):
```

Add all items in the Random node to a group:

```
grp = addgroup(da, 'ServerItemsEx');
itm = additem(grp, serveritems(da, 'Random.*'));
```

Version History

Introduced before R2006a

See Also

`getnamespace` | `serveritemprops`

set

Configure or display OPC object properties

Syntax

```
set(Obj)
Prop = set(Obj)
set(Obj,'PropertyName')
Prop = set(Obj,'PropertyName')
set(Obj,'PropertyName',PropertyValue)
set(Obj,S)
set(Obj,PN,PV)
set(Obj,'PropName1',PropValue1,'PropName2',PropValue2,...)
```

Description

`set(Obj)` displays property names and any enumerated values for all configurable properties of OPC object `Obj`. `Obj` must be a single toolbox object.

`Prop = set(Obj)` returns all property names and their possible values for object `Obj`. `Obj` must be a single object. The return value, `Prop`, is a structure whose field names are the property names of `Obj`, and whose values are cell arrays of possible property values or empty cell arrays if the property does not have a finite set of possible character vector values.

`set(Obj,'PropertyName')` displays the possible values for the specified property, `PropertyName`, of toolbox object `Obj`. `Obj` must be a single object.

`Prop = set(Obj,'PropertyName')` returns the possible values for the specified property, `PropertyName`, of object `Obj`. The returned array, `Prop`, is a cell array of possible value character vectors or an empty cell array if the property does not have a finite set of possible character vector values.

`set(Obj,'PropertyName',PropertyValue)` sets the value, `PropertyValue`, of the specified property, `PropertyName`, for object `Obj`. `Obj` can be a vector of toolbox objects, in which case `set` sets the property values for all the objects specified.

Note that if `Obj` is connected to an OPC server, configuring server-specific properties such as `UpdateRate` and `DeadbandPercent` might be time consuming.

`set(Obj,S)` where `S` is a structure whose field names are object property names, sets the properties named in each field name to the values contained in the structure.

`set(Obj,PN,PV)` sets the properties specified in the cell array of character vectors or string array, `PN`, to the corresponding values in the cell array `PV`, for all objects specified in `Obj`. The cell array `PN` must be a vector, but the cell array `PV` can be `M`-by-`N`, where `M` is equal to `length(Obj)` and `N` is equal to `length(PN)`, so that each object will be updated with a different set of values for the list of property names contained in `PN`.

`set(Obj,'PropName1',PropValue1,'PropName2',PropValue2,...)` sets multiple property values with a single statement.

Note that it is permissible to use name-value pairs, structures, and name-value cell array pairs in the same call to `set`.

Examples

Create an `opcda` object and add a group to that object.

```
da = opcda('localhost','Dummy.Server');
grp = addgroup(da,'SetExample');
```

Set the `opcda` object's `Timeout` to `300` seconds, and restrict the event log to `2000` entries.

```
set(da,'Timeout',300,'EventLogMax',2000);
```

Set multiple properties using cell array pairs.

```
set(da,{'Name','ServerID'},{'My Opcda object','OPC.Server.1'});
```

Set the group name.

```
set(grp,'Name','myopcgroup');
```

Query the permissible values for the group's `Subscription` property.

```
set(grp,'Subscription')
```

Tips

As an alternative to the `set` function, you can directly assign property values using dot-notation. The following two lines achieve the same result.

```
set(daObj,'Timeout',10);
daObj.Timeout = 10;
```

Version History

Introduced before R2006a

See Also

Functions
`get` | `opchelp` | `propinfo`

setSecurityModel

Package: `opc.ua`

Set security configuration parameters for OPC UA client

Syntax

```
setSecurityModel(UaClient,'Best')
setSecurityModel(UaClient,MessageMode,ChannelPolicy)
```

Description

`setSecurityModel(UaClient,'Best')` sets both the `MessageSecurityMode` and `ChannelSecurityPolicy` properties of the OPC UA client `UaClient` to the best possible security configuration available for the server. The client attempts to retrieve available endpoints from the server if those are not yet retrieved.

`setSecurityModel(UaClient,MessageMode,ChannelPolicy)` sets the `MessageSecurityMode` and `ChannelSecurityPolicy` properties of OPC UA client `UaClient` to the specified `MessageMode` and `ChannelPolicy`, respectively. If a matching endpoint cannot be found in the list of known endpoints, an error occurs.

Examples

Set Security Mode

Set the OPC UA client security mode for signed but not encrypted messages.

```
s = opcuaserverinfo('localhost');
UaClient = opcua(s);
setSecurityModel(UaClient,'Sign');
connect(UaClient);
```

Input Arguments

`UaClient` — OPC UA client
`opc.ua.Client` object

OPC UA client, specified as an `opc.ua.Client` object. You can create the client using the `opcua` function.

Example: `opcua()`

`MessageMode` — Client message security mode
`'None'` | `'Sign'` | `'SignAndEncrypt'`

Client message security mode, specified as a character vector or string. Either `MessageMode` or `ChannelPolicy` can be empty, but not both. In this case, the highest security model is chosen from the available endpoints to match the given option.

Example: `'Sign'`

Data Types: `char` | `string`

`ChannelPolicy` — Client channel security policy
char | string

Client channel security policy, specified as a character vector or string.

`ChannelPolicy` must be specified as one of the enumerations defined in `opc.ua.ChannelSecurityPolicies`. For example,

```
enumeration opc.ua.ChannelSecurityPolicies
```

```
Enumeration members for class 'opc.ua.ChannelSecurityPolicies':

    Unknown
    None
    Aes128_Sha256_RsaOaep
    Basic256Sha256
    Aes256_Sha256_RsaPss
```

Example: `'Basic256Sha256'`

Data Types: `char` | `string`

Version History

Introduced in R2020a

R2023a: Basic128Rsa15 and Basic256 Policy Support Being Removed
Warns starting in R2023a

Support for the `Basic128Rsa15` and `Basic256` security policies will be removed in a future release. Consider using one of the following security policies instead: `Aes128_Sha256_RsaOaep`, `Aes256_Sha256_RsaPss`, or `Basic256Sha256`.

See Also

Functions
`opcua` | `connect (opcua)` | `exportClientCertificate`

Topics
"OPC UA Security" on page 17-7
"OPC UA Certificate Management" on page 17-9

showopcevents

Event log summary for OPC events

Syntax

```
showopcevents(DAObj)
showopcevents(DAObj,Index)
showopcevents(Struct)
showopcevents(Struct,Index)
```

Description

`showopcevents(DAObj)` displays a summary of the event log for the `opcda` object specified by `DAObj`.

`showopcevents(DAObj,Index)` displays a summary of the events with index of `Index`. `Index` can be the numerical index, a character vector, or a cell array of character vectors that specifies the type of event. Valid events are `CancelAsync`, `Error`, `ReadAsync`, `Shutdown`, `Start`, `Stop`, and `WriteAsync`.

`showopcevents(Struct)` and `showopcevents(Struct,Index)` display a summary of the events with index of `Index` for the event structure, `Struct`. You can obtain an event structure from the object's `EventLog` property.

The display summary includes the event type, the local time the event occurred, and additional event-specific information.

Examples

Configure a logging task for the Matrikon Simulation Server, then display the event log to find timing information for the logging task:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da)
grp = addgroup(da);
grp.RecordsToAcquire = 10;
itm = additem(grp,'Bucket Brigade.Real8');
start(grp);
wait(grp);
showopcevents(da);
```

Version History

Introduced before R2006a

See Also

`opccallback`

showValues

Class: `opc.hda.Data`
Package: `opc.hda`

Display table of values for OPC HDA data object

Syntax

`showValues(dObj)`

Description

`showValues(dObj)` displays a table of values for OPC HDA object `hdaObj`. If `hdaObj` is a scalar object, the table lists each time stamp with its corresponding value and quality.

If `hdaObj` is an array with all items having the same time stamps, the table shows the time stamp followed by each item's value.

If `hdaObj` is an array with items having different time stamps, an error is generated. Use the `tsunion` method to generate an array with each item containing the same time stamps.

The date format for the time stamps is controlled by the OPC date display preference, which you can set by using `opc.setDateDisplayFormat`.

Examples

Load the OPC HDA example data file and show the values of the first `hdaDataSmall` object:

```
load opcSampleHdaData;
showValues(hdaDataSmall(1))
```

See Also

`disp`

single

Package: `opc.hda`

Convert OPC HDA data object array to single matrix

Syntax

`Vsingle = single(DObj)`

Description

`Vsingle = single(DObj)` converts the OPC HDA data object array `DObj` into a matrix of data type `single`.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to Matrix of singles

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create a matrix of type `single` from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vsingle = single(dUnion);
```

Input Arguments

`DObj` — OPC HDA data
OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vsingle` — OPC HDA data values
matrix of `single` type

OPC HDA data values, returned as a matrix of `single` type. `Vsingle` is constructed as an `M`-by-`N` matrix of `single` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

stairs

Package: `opc`

Plot OPC HDA data object as stairstep graph

Syntax

```
stairs(dObj)
pH = stairs(dObj)
```

Description

`stairs(dObj)` plots the data in OPC HDA data object `dObj` as a series of stair steps. Each element of `dObj` is plotted into the current axes as the value against its time stamp. Quality is not displayed in the plot.

`pH = stairs(dObj)` returns the handles to the created stairseries objects in `pH`.

In all cases, if the current plot is not held, the X-axis is updated using `datetick` to show date ticks instead of numeric ticks.

Examples

Load the OPC HDA example data file and plot the `hdaDataVis` object as a stairstep graph:

```
load opcSampleHdaData;
stairs(hdaDataVis)
```

See Also

`datetick` | `plot` | `stairs`

start

Start a logging task

Syntax

```
start(GObj)
```

Description

`start(GObj)` starts a data logging task for `GObj`. `GObj` can be a scalar `dagroup` object, or a vector of `dagroup` objects. A `dagroup` object must be `active` and contain at least one item for `start` to succeed.

When logging is started, `GObj` performs the following operations:

1. Generates a `Start` event, and executes the `StartFcn` callback.
2. If `Subscription` is `'off'`, sets `Subscription` to `'on'` and issues a warning.
3. Removes all records associated with the object from the toolbox engine.
4. Sets `RecordsAcquired` and `RecordsAvailable` to `0`.
5. Sets the `Logging` property to `'on'`.

The `Start` event is logged to the `EventLog`.

`GObj` will stop logging when a `stop` command is issued, or when `RecordsAcquired` reaches `RecordsToAcquire`.

Examples

Configure and start a logging task for 30 seconds of data:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'StartEx');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-toothed Waves.UInt16');
grp.LoggingMode = 'memory';
grp.UpdateRate = 0.5;
grp.RecordsToAcquire = 60;
start(grp);
```

Wait for the logging task to finish, then retrieve the records into a `double` array and plot the data with a legend:

```
wait(grp);
[itmID, val, qual, tStamp] = getdata(grp, 'double');
plot(tStamp(:,1), val(:,1), tStamp(:,2), val(:,2));
legend(itmID);
datetick x keeplimits
```

Version History

Introduced before R2006a

See Also

`flushdata` | `getdata` | `peekdata` | `stop` | `wait`

stop

Stop a logging task

Syntax

```
stop(GObj)
```

Description

`stop(GObj)` stops all logging tasks associated with the `dagroup` object `GObj`. `GObj` can be a `dagroup` object or a vector of `dagroup` objects. When the function stops a logging task, it sets the object's `Logging` property value to `'Off'`, and triggers execution of the object's `StopFcn` callback.

A `dagroup` object also stops running when the logging task has acquired all the requested records. This occurs when `RecordsAcquired` equals `RecordsToAcquire`.

The object's `EventLog` property records the `Stop` event.

Examples

Configure and start a logging task for 30 seconds of data:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExOPCREAD');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-Toothed Waves.Int2');
grp.LoggingMode = 'memory';
grp.UpdateRate = 0.5;
grp.RecordsToAcquire = 60;
start(grp);
```

Stop the logging task after 5 seconds:

```
wait(5);
stop(grp);
```

Version History

Introduced before R2006a

See Also

`start` | `wait`

subscribe

Package: `icomm.mqtt`

Subscribe to MQTT topic

Syntax

```
SubscrTable = subscribe(mqttClient,mqttTopic)
SubscrTable = subscribe(mqttClient,mqttTopic,Name=Value)
```

Description

`SubscrTable = subscribe(mqttClient,mqttTopic)` subscribes the MQTT client to the specified MQTT topic, and returns a table listing all subscriptions for that client.

`SubscrTable = subscribe(mqttClient,mqttTopic,Name=Value)` specifies additional subscription behaviors using optional name-value pairs.

Examples

Subscribe to MQTT Topics

Create an MQTT client connected to the Eclipse HiveMQ™ public broker and subscribe to several topics with different options.

Subscribe to the topic `"trubits/mqTop48"`.

```
mqttClient = mqttclient("tcp://broker.hivemq.com");
mySub = subscribe(mqttClient,"trubits/mqTop48")
```

```
mySub = 

  1×3 table

          Topic          QualityOfService    Callback
    _________________    ________________    ________

    "trubits/mqTop48"           0               ""
```

```
mySub = subscribe(mqttClient, "trubits/mqTmp52",QualityOfService=2)
```

```
mySub = 

  2×3 table

          Topic          QualityOfService    Callback
    _________________    ________________    ________

    "trubits/mqTop48"           0               ""
    "trubits/mqTmp52"           2               ""
```

Create a callback function in the file `showmessage.m` that displays the topic and received message.

```
function showMessage(topic,data)
    disp(topic);
    disp(data);
end
```

Subscribe to a topic to have the callback executed when a message is received.

```
mySub = subscribe(mqttClient,"trubits/mqTsp61",Callback=@showmessage)
```

```
mySub = 

  3×3 table

          Topic            QualityOfService      Callback    
    _________________    ________________    _____________

    "trubits/mqTop48"           0            ""           
    "trubits/mqTmp52"           2            ""           
    "trubits/mqTsp61"           0            "showmessage"
```

Input Arguments

`mqttClient` — MQTT client

`Client` object

MQTT client specified as an `icomm.mqtt.Client` object, created with the `mqttclient` function.

Example: `mqttClient = mqttclient()`

Data Types: `object`

`mqttTopic` — MQTT topic

string | char

MQTT topic to subscribe to, specified as a string or character vector.

Example: `"trubits/mqTop48"`

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Callback="showMsg"`

`QualityOfService` — Quality of Service (QoS)

0 (default) | 1 | 2

Quality of Service (QoS) for message delivery, specified as an integer value of `0`, `1`, or `2`:

- `0` — Messages delivered at most once, not more (default).
- `1` — Messages delivered at least once, not less.
- `2` — Messages delivered exactly once, not more or less.

Example: `QualityOfService=1`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Callback` — Function to execute when message received
function handle | string | char

Function to execute when a message is received on the subscribed topic, specified as a function handle, string, or character vector. The function is called with two input arguments (topic, data) in that order.

Example: `Callback=@notifyUser`

Data Types: `char` | `string` | `function_handle`

Output Arguments

`SubscrTable` — Subscriptions table
table

Table of all topics that the MQTT client is subscribed to.

Version History

Introduced in R2022a

See Also

Functions
`mqttclient` | `read` | `unsubscribe` | `peek` | `flush`

tags

Package: `icomm.pi`

List tags from OSIsoft PI server

Syntax

```
tagList = tags(piClient)
tagList = tags(piClient,Name=tagName)
```

Description

`tagList = tags(piClient)` returns a list of all tags available from the OSIsoft PI that the client `piCient` is connected to. A tag is used by the PI system as an alias or shortcut to represent an asset attribute such as voltage, current, temperature, etc. Some tag names are short, others might be long and descriptive or include a unique ID.

`tagList = tags(piClient,Name=tagName)` filters the list of available tags to match the name specified by `tagName`. The wildcard character * is supported for partial string or pattern matching.

Examples

Find Tags on an OSIsoft PI Server

Request a list of all tags from the server.

```
tagList = tags(piClient);
```

Request a list of all tags from the server containing a matching string.

```
tagList = tags(piClient,"*pressure*");
```

Input Arguments

`piClient` — Client connected to OSIsoft PI server
`icomm.pi.Client` object

Client connected to OSIsoft PI server, specified as an `icomm.pi.Client` object. You create the object with the `piclient` function.

Example: `piClient = piclient(_)`

Data Types: `object`

`tagName` — Tag name to match on
string | char

Tag name to match on, specified as a string or character vector. You can use the wildcard * character to perform partial matching. For example:

```
"tagName"    matches only the exact name tagName.
"tagName*"   matches tags that start with tagName.
"*tagName*"  matches tags containing tagName anywhere in their name.
```

Example: `"Power*"`

Data Types: `char` | `string`

Output Arguments

`tagList` — List of tags from OSIsoft PI server
table

List of tags from OSIsoft PI server, returned as an N-by-1 table of strings with one tag per row.

Version History
Introduced in R2022a

See Also

Functions
`piclient` | `read`

trend

Display graphical trend of OPC data for group

Syntax

```
H = trend(GObj)
H = trend(GObj, 'PropertyName', PropertyValue,...)
```

Description

`H = trend(GObj)` displays the newest 100 points of live data for the items defined in the `dagroup` object `GObj` in the current axes. `GObj` must be an active group containing one or more items. The handles to the created Handle Graphics® objects are returned in `H`.

All the items are displayed in the same axes, with no scaling. New data is displayed on the far right of the axes, and oldest data is displayed on the left. If no old data exists (such as at the beginning of a plot), the axes are empty. The Handle Graphics objects (including the axis limits) are updated with new data whenever the group object receives a Data Change event from the OPC server.

`H = trend(GObj, 'PropertyName', PropertyValue,...)` allows you to pass additional property/value pairs to specify additional properties of the created plots. Special property/value pairs are listed in the following table. If any property is not in this list, that property/value pair is passed on to the created Handle Graphics objects.

Property Name	Description	Default
`DisplayTime`	Defines the number of seconds of history to display in the plot.	`100*gObj.UpdateRate`
`Parent`	Defines the parent axes objects in which to display the trends. The value can be a scalar, or a vector the same length as the number of items in `GObj`. If the value is a vector, each item's value is displayed in the respective axes object.	Current axes
`PlotType`	Defines the plot types for each item. Valid plot types are `'line'`, `'stairs'`, and `'stem'`. The value can be a scalar, or a cell array the same length as the number of items in `GObj`. If the value is a cell array of character vectors, each item's plot type is set to the respective plot type in the value array.	`'line'`
`DateTimeFormat`	Sets the display format for the *x*-axis of all axes objects into which data is plotted. `DateTimeFormat` must be one of the date formats recognized by `datetime`.	Depends on system locale. See "Set Date and Time Display Format".

Property Name	Description	Default
`BufferTime`	Defines the number of seconds of history to store for all items. Setting this value to a number greater than the value of `DisplayTime` allows you to pause the trend (by setting the `Subscription` property of the group to `'off'`) and panning the axes in question.	`10*DisplayTime`

You can fix the axes *y*-limits to a particular value by using the `YLim` property of the axes containing your visualized data. For example, to set the limits of the *y*-axis to the instrument range reported by the OPC server, use the following code:

```
props = serveritemprops(da,itmName,102:103);
currentAxes = gca;
currentAxes.YLim = [props.PropValue];
```

If you add items to a group that currently has an active trend, the item is not shown. Call `trend` again to include that item in the trend view. (If you set the hold state of the axes to `'on'`, when you call `trend`, existing trend objects are reused, without destroying their current view.)

If you delete an item from a group that currently has an active trend, the trend display shows no data for that item, and the item's trend eventually disappears off the graph.

This function overwrites the following properties of the group object:

- The DataChangeFcn property is set to update the axes with new data whenever it is received from the OPC server. If there is an existing `DataChangeFcn` callback, the trend functionality overwrites the callback.
- The Subscription property is configured to `'on'` to receive Data Change events from the OPC server. You can change `Subscription` to `'off'` after calling `trend`, in which case the trend stops updating until you set `Subscription` back to `'on'` or issue a `readasync` command.

Examples

Configure a group with two items:

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExOPCTREND');
itm1 = additem(grp,'Triangle Waves.Real8');
itm2 = additem(grp,'Saw-Toothed Waves.Int2');
```

Create a trend showing the last two minutes of data in two separate axes:

```
ax1 = subplot(2,1,1);
ax2 = subplot(2,1,2);
trend(grp,'DisplayTime',120,'Parent',[ax1,ax2]);
```

Version History

Introduced in R2007b

R2023a: `DateFormat` Argument Being Removed
Warns starting in R2023a

The `DateFormat` name-value argument will be removed in a future release. Use `DateTimeFormat` instead.

See Also

Functions
`datetime` | `hold`

tsintersect

Class: `opc.hda.Data`
Package: `opc.hda`

Intersection of time stamp in OPC HDA data object

Syntax

```
NewObj = tsintersect(DObj)
```

Description

`NewObj = tsintersect(DObj)` resamples data in OPC HDA data object `DObj` so that all elements of the object have the same time stamps given by the intersection of all time stamps in all elements of `DObj`.

If `DObj` contains elements with the same item ID, those elements are combined into one element. So the size of `NewObj` might be smaller than the size of `DObj`.

Examples

Load the OPC HDA example data file and find all common values of `hdaDataSmall`:

```
load opcSampleHdaData;
newObj = tsintersect(hdaDataSmall);
```

Display the values and qualities of the new object:

```
newObj.showValues
```

See Also

`resample` | `tsunion` | `showValues`

tsunion

Class: `opc.hda.Data`
Package: `opc.hda`

Union of time stamps in an OPC HDA data object

Syntax

```
NewObj = tsunion(DObj)
NewObj = tsunion(DObj,'linear')
NewObj = tsunion(DObj,'hold')
NewObj = tsunion(DObj,'nearest')
NewObj = tsunion(DObj,'spline')
NewObj = tsunion(DObj,'pchip')
```

Description

`NewObj = tsunion(DObj)` merges the time stamps of all items (elements) in data object `DObj`, so that each element of `NewObj` has the same time stamp vector corresponding to all possible time stamps in all elements of `DObj`. For each element, values are linearly interpolated or extrapolated where that time stamp does not exist for an item (element of the Data object).

If `DObj` contains elements with the same item ID, those elements are combined into one element. So the size of `NewObj` might be smaller than the size of `DObj`.

Quality for the resampled data is set as follows:

- All original values retain their quality.
- All interpolated values get a quality of `Interpolated: Good`.
- All extrapolated values get a quality of `Interpolated: Sub-Normal`.

`NewObj = tsunion(DObj,'linear')` uses linear interpolation.

`NewObj = tsunion(DObj,'hold')` uses a zero-order hold interpolation where the previous known value is used for all new time stamps. Any time stamp prior to the first known value is set to `NaN` (or `0` if the value is a fixed-point data type).

`NewObj = tsunion(DObj,'nearest')` uses nearest-neighbor interpolation as defined by `interp1`.

`NewObj = tsunion(DObj,'spline')` uses spline interpolation as defined by `interp1`.

`NewObj = tsunion(DObj,'pchip')` uses shape-preserving, piece-wise, cubic interpolation as defined by `interp1`.

For data objects containing character vector values, only the `'hold'` method can be used.

Examples

Load the OPC HDA example data file and find the time stamp union of `hdaDataSmall`:

```
load opcSampleHdaData;
newObj = tsunion(hdaDataSmall);
```

Find the union using `'hold'` resampling:

```
newObjHold = tsunion(hdaDataSmall, 'hold');
```

See Also

`interp1` | `resample` | `showValues` | `tsintersect`

uint16

Package: `opc.hda`

Convert OPC HDA data object array to uint16 matrix

Syntax

```
Vuint16 = uint16(DObj)
```

Description

`Vuint16 = uint16(DObj)` converts the OPC HDA data object array `DObj` into a `uint16` matrix.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to uint16 Matrix

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create a `uint16` matrix from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vuint16 = uint16(dUnion);
```

Input Arguments

`DObj` — OPC HDA data

OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vuint16` — OPC HDA data values

`uint16` matrix

OPC HDA data values, returned as a `uint16` matrix. `Vuint16` is constructed as an `M`-by-`N` matrix of `uint16` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

uint32

Package: `opc.hda`

Convert OPC HDA data object array to uint32 matrix

Syntax

`Vuint32 = uint32(DObj)`

Description

`Vuint32 = uint32(DObj)` converts the OPC HDA data object array `DObj` into a `uint32` matrix.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to uint32 Matrix

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create a `uint32` matrix from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vuint32 = uint32(dUnion);
```

Input Arguments

`DObj` — OPC HDA data

OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vuint32` — OPC HDA data values

`uint32` matrix

OPC HDA data values, returned as a `uint32` matrix. `Vuint32` is constructed as an `M`-by-`N` matrix of `uint32` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

uint64

Package: opc.hda

Convert OPC HDA data object array to uint64 matrix

Syntax

```
Vuint64 = uint64(DObj)
```

Description

`Vuint64 = uint64(DObj)` converts the OPC HDA data object array `DObj` into a `uint64` matrix.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to uint64 Matrix

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create a `uint64` matrix from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vuint64 = uint64(dUnion);
```

Input Arguments

`DObj` — OPC HDA data

OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vuint64` — OPC HDA data values

`uint64` matrix

OPC HDA data values, returned as a `uint64` matrix. `Vuint64` is constructed as an `M`-by-`N` matrix of `uint64` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

uint8

Package: `opc.hda`

Convert OPC HDA data object array to uint8 matrix

Syntax

```
Vuint8 = uint8(DObj)
```

Description

`Vuint8 = uint8(DObj)` converts the OPC HDA data object array `DObj` into a `uint8` matrix.

`DObj` must have the same time stamps for each of the Item IDs (elements of `DObj`), otherwise an error is generated. Use `tsunion`, `tsintersect`, or `resample` to generate an OPC HDA data object containing the same time stamp for all items in the object.

Examples

Convert OPC HDA Data to uint8 Matrix

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create a `uint8` matrix from the result.

```
load opcSampleHdaData;
dUnion = tsunion(hdaDataSmall);
Vuint8 = uint8(dUnion);
```

Input Arguments

`DObj` — OPC HDA data
OPC HDA data object array

OPC HDA data, specified as an OPC HDA data object array.

Output Arguments

`Vuint8` — OPC HDA data values
`uint8` matrix

OPC HDA data values, returned as a `uint8` matrix. `Vuint8` is constructed as an `M`-by-`N` matrix of `uint8` values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

Version History

Introduced in R2011a

See Also

Functions
`resample` | `tsintersect` | `tsunion`

unsubscribe

Package: `icomm.mqtt`

Unsubscribe from MQTT topics

Syntax

```
unsubscribe(mqttClient)
unsubscribe(mqttClient,Topic=mqttTopic)
```

Description

`unsubscribe(mqttClient)` unsubscribes the MQTT client from all its subscribed topics.

`unsubscribe(mqttClient,Topic=mqttTopic)` unsubscribes the MQTT client from the specified topic, `mqttTopic`.

Examples

Unsubscribe from MQTT Topics

Unsubcribe a client from one, then from all MQTT topics.

View the subscriptions of a client.

```
mqttClient.Subscriptions
```

```
ans =

  3×3 table

          Topic          QualityOfService      Callback
    _________________    ________________    _____________

    "trubits/mqTop48"           0            ""
    "trubits/mqTmp52"           2            ""
    "trubits/mqTsp61"           0            "showmessage"
```

Unsubscribe from one topic.

```
unsubscribe(mqttClient,Topic="trubits/mqTsp61")
mqttClient.Subscriptions
```

```
ans =

  2×3 table

          Topic          QualityOfService    Callback
    _________________    ________________    ________

    "trubits/mqTop48"           0               ""
    "trubits/mqTmp52"           2               ""
```

Unsubscribe from all remaining topics.

```
unsubscribe(mqttClient)
mqttClient.Subscriptions
```

```
ans = 

  0×3 empty table
```

Input Arguments

`mqttClient` — MQTT client
`Client` object

MQTT client specified as an `icomm.mqtt.Client` object, created with the `mqttclient` function.

Example: `mqttClient = mqttclient()`

Data Types: `object`

`mqttTopic` — MQTT topic
string | char

MQTT topic to unsubscribe from, specified as a string or character vector.

Example: `"trubits/mqTop48"`

Data Types: `string` | `char`

Version History

Introduced in R2022a

See Also

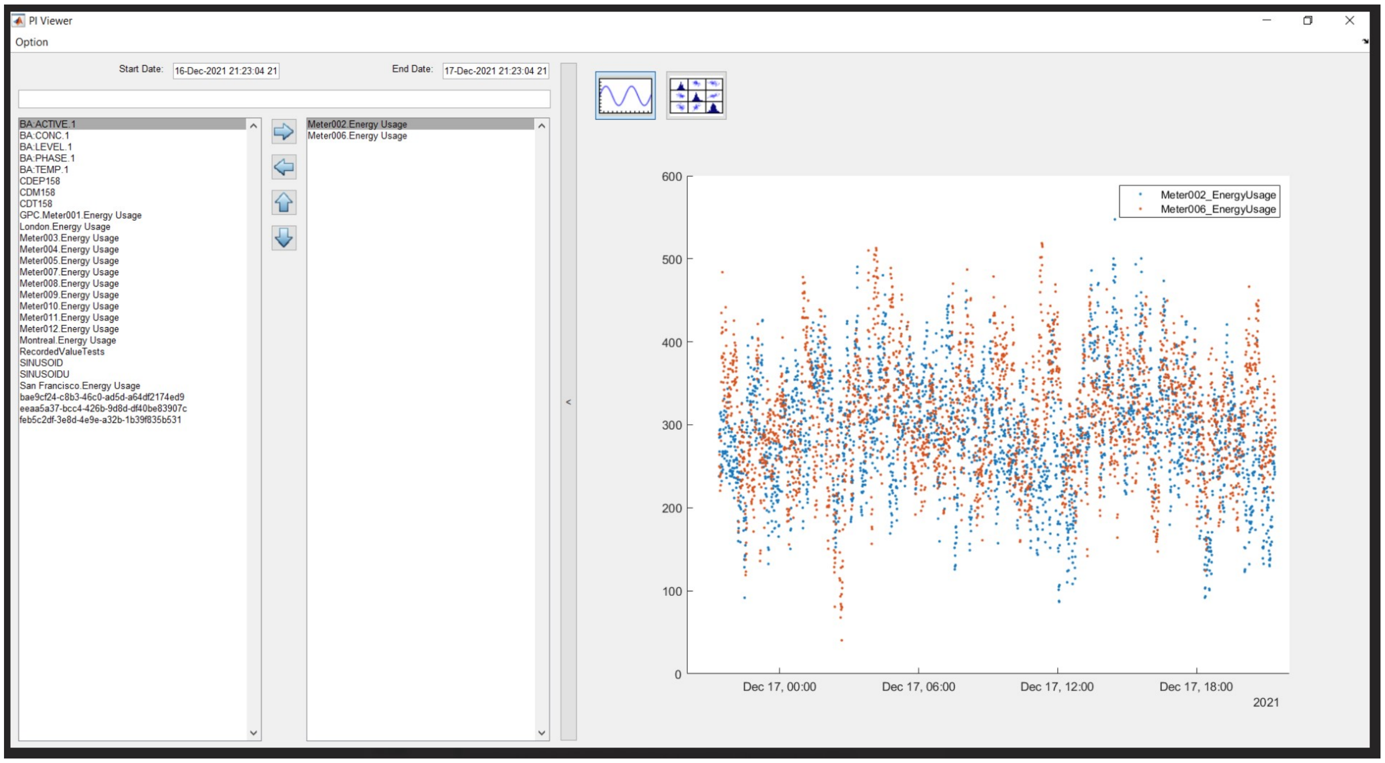
Functions
`mqttclient` | `subscribe`

PI Viewer

Visualize data from OSIsoft PI Server

Description

The **PI Viewer** allows you to graphically search and select tags on an OSIsoft PI Server, then plot data from those tags.



Start Date: 16-Dec-2021 21:23:04 21
End Date: 17-Dec-2021 21:23:04 21

BA:ACTIVE.1
BA:CONC.1
BA:LEVEL.1
BA:PHASE.1
BA:TEMP.1
CDEP158
CDM158
CDT158
GPC.Meter001.Energy Usage
London.Energy Usage
Meter003.Energy Usage
Meter004.Energy Usage
Meter005.Energy Usage
Meter007.Energy Usage
Meter008.Energy Usage
Meter009.Energy Usage
Meter010.Energy Usage
Meter011.Energy Usage
Meter012.Energy Usage
Montreal.Energy Usage
RecordedValueTests
SINUSOID
SINUSOIDU
San Francisco.Energy Usage
bae9cf24-c8b3-46c0-ad5d-a64df2174ed9
eeaa5a37-bcc4-426b-9d8d-df40be83907c
feb5c2df-3e8d-4e9e-a32b-1b39f835b531

Meter002.Energy Usage
Meter006.Energy Usage

Open the PI Viewer App

To open the **PI Viewer** app, at the MATLAB command line type:

```
viewer(piClient)
```

where `piClient` is the OSIsoft PI client created with the `piclient` function.

Examples

View Data from OSIsoft PI Server Tags

Create a `piclient` object and open the **PI Viewer** for reading tags from the OSIsoft PI Server.

```
piClient = piclient("pi-host-55");
viewer(piClient)
```

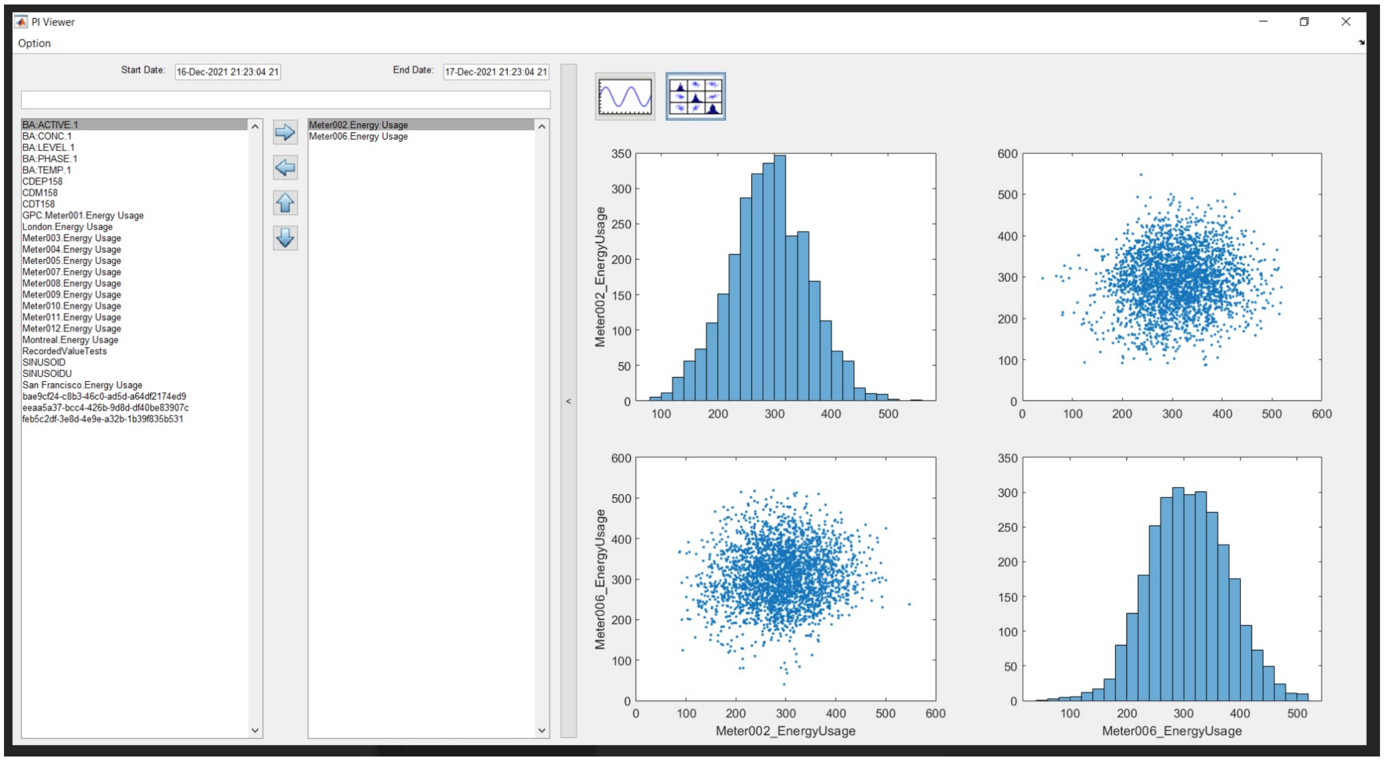
Select the tags you want to read from, and click the right-arrow to add them to the right-hand column.

If you want to limit the date range, click the **Start Date** field and **End Date** field. A calendar pops up for you to select the start and stop dates.

Select the type of graphs you want to view:

- Click to plot all data points on a common time axis.
- Click to generate a matrix of plots, displaying histograms and scatter plots by data groups, in a manner similar to the `gplotmatrix` function.

  The **PI Viewer** display might look something like this:



Start Date: 16-Dec-2021 21:23:04 21

End Date: 17-Dec-2021 21:23:04 21

```
BA:ACTIVE.1
BA:CONC.1
BA:LEVEL.1
BA:PHASE.1
BA:TEMP.1
CDEP158
CDM158
CDT158
GPC.Meter001.Energy Usage
London.Energy Usage
Meter003.Energy Usage
Meter004.Energy Usage
Meter005.Energy Usage
Meter007.Energy Usage
Meter008.Energy Usage
Meter009.Energy Usage
Meter010.Energy Usage
Meter011.Energy Usage
Meter012.Energy Usage
Montreal.Energy Usage
RecordedValueTests
SINUSOID
SINUSOIDU
San Francisco.Energy Usage
bae9cf24-c8b3-46c0-ad5d-a64df2174ed9
eeaa5a37-bcc4-426b-9d8d-df40be83907c
feb5c2df-3e8d-4e9e-a32b-1b39f835b531
```

```
Meter002.Energy Usage
Meter006.Energy Usage
```

Version History

Introduced in R2022a

See Also

Functions

`piclient` | `tags` | `read`

wait

Suspend MATLAB execution until object stops logging

Syntax

```
wait(GObj)
wait(GObj,TSec)
```

Description

`wait(GObj)` suspends MATLAB execution until the group object `GObj` has stopped logging. `GObj` must be a scalar `dagroup` object.

Use the `wait` function when you want to guarantee that all data is logged before another task is performed.

You can press `Ctrl+C` to interrupt the `wait` function. An error message appears, and control returns to the MATLAB command window.

`wait(GObj,TSec)` waits at most `TSec` seconds for `GObj` to stop logging. If the group object is still logging when the timeout maximum value is exceeded, an error is generated.

Examples

Wait for Logging to Complete Before Plotting

Log 60 seconds of data and plot the results.

Log 60 seconds of data at 1-second intervals from the Matrikon simulation server tags `Random.Real8` and `Random.UInt4`. When logging is complete, display a message, then retrieve and plot the data.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da)
grp = addgroup(da,'WaitExample');
itm = additem(grp,{'Random.Real8','Random.UInt4'});
grp.RecordsToAcquire = 60;
grp.UpdateRate = 1;
start(grp);
wait(grp)
disp('Acquisition complete.')
[itmID,v,q,t]=getdata(grp,'double');
plot(t(:,1),v(:,1),t(:,2),v(:,2));
legend(itmID);
```

Input Arguments

`GObj` — OPC DA group
DA group object

OPC DA group, specified as a DA group object.

Example: `addgroup()`

TSec — Maximum wait time
seconds

Maximum wait time, specified as seconds in any numeric type.

Example: `60`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Version History
Introduced before R2006a

See Also

Functions
`getdata` | `start` | `stop`

write

Write values to group or items

Syntax

```
write(GObj,Values)
write(IObj,Values)
```

Description

`write(GObj,Values)` writes values to all the items contained in the `dagroup` object `GObj`. `Values` is a cell array of values--one for each item. To ensure that a specific value is written to the correct item object, you should construct the `Values` cell array based on the order of the items returned by the `Item` property of `GObj`.

`write(IObj,Values)` writes values to all the items contained in the vector of `daitem` objects specified by `IObj`.

The data types of the values do not need to match the canonical data type of the associated items. However an error is returned if a data type conversion cannot be performed.

Because the values are written to the device, this operation might be slow. The function does not return until it verifies that the device has actually accepted or rejected the data.

Note The behavior of an OPC server when writing `NaN` to an item is server-dependent. If you attempt to write `NaN` to an OPC server, the value might be silently ignored by the OPC server. That is, the server might not generate any events in response to writing `NaN` to an item.

Examples

Write Values to OPC DA Items

Configure a client, group, and items for the Matrikon Simulation Server.

```
da = opcda('localhost','Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da,'ExWrite');
itm = additem(grp,{'Bucket Brigade.Real8', ...
    'Bucket Brigade.String'});
```

Write and read values to and from the items.

```
write(grp,{23,'Hello World!'})
r1 = read(grp);
```

```
write(itm(1),15)
r2 = read(itm(1));
```

Input Arguments

GObj — OPC DA group
dagroup

OPC DA group, specified as a dagroup object.

Example: addgroup()

IObj — OPC DA items
daitem

OPC DA items, specified as an array of daitem objects.

Example: additem()

Values — Data values
cell

Data values, specified as a cell array.

Example: {23,'Label4'}

Data Types: cell

Version History

Introduced before R2006a

See Also

Functions
read | readasync | refresh | writeasync

write

Package: `icomm.mqtt`

Write message to MQTT topic

Syntax

```
write(mqttClient,mqttTopic,mqttMsg)
write(mqttClient,mqttTopic,mqttMsg,Name=Value)
```

Description

`write(mqttClient,mqttTopic,mqttMsg)` writes the message string `mqttMsg` to the topic `mqttTopic` from the connected client `mqttClient`.

`write(mqttClient,mqttTopic,mqttMsg,Name=Value)` uses additional options specified by one or more name-value pair arguments.

Examples

Write Messages to an MQTT Topic

Create an MQTT client and write messages to a topic with various options.

Create an MQTT client connected to the Eclipse HiveMQ™ public broker and write a message to the topic `myTopic`.

```
mqttClient = mqttclient("tcp://broker.hivemq.com");
write(mqttClient,"myTopic","Hello World")
```

Write to the topic with QualityOfService 2.

```
write(mqttClient,"myTopic","High Service Message",QualityOfService=2)
```

Write a message to be retained by the broker.

```
write(mqttClient,"myTopic","Msg for new subscribers",Retain=true)
```

Input Arguments

`mqttClient` — MQTT client
`Client` object

MQTT client specified as an `icomm.mqtt.Client` object, created with the `mqttclient` function.

Example: `mqttClient = mqttclient()`

Data Types: `object`

`mqttTopic` — MQTT topic
string | char

MQTT topic to write message to, specified as a string or character vector.

Example: `"trubits/mqTop48"`

Data Types: `string` | `char`

`mqttMsg` — MQTT message to write to topic
string | char

MQTT message to write to the topic, specified as a string or character vector.

Example: `"Hello World"`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `QualityOfService=1`

`QualityOfService` — Quality of Service (QoS)
0 (default) | 1 | 2

Quality of Service (QoS) for message delivery, specified as an integer value of `0`, `1`, or `2`:

- `0` — Messages delivered at most once, not more (default).
- `1` — Messages delivered at least once, not less.
- `2` — Messages delivered exactly once, not more or less.

Example: `QualityOfService=1`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Retain` — Specify whether broke retains message
`false` (default) | `true`

Control whether the broker retains the message for any new subscriber to the topic, specified as a logical `false` (default) or `true`.

Example: `Retain=true`

Data Types: `logical`

Version History

Introduced in R2022a

See Also

Functions
`mqttclient` | `subscribe` | `read` | `peek`

write

Perform a write operation to the connected Modbus server

Syntax

```
write(m,target,address,values)
write(m,target,address,values,serverId)
write(m,'holdingregs',address,values,'precision')
write(m,'holdingregs',address,values,serverId,'precision')
```

Description

`write(m,target,address,values)` writes data to Modbus object `m` to target type `target` at the starting address `address` using the values to write `values`. You can write to coils or holding registers.

`write(m,target,address,values,serverId)` additionally specifies `serverId`, which is the address of the server to send the write command to. `serverId` can be used for coils or holding registers.

`write(m,'holdingregs',address,values,'precision')` additionally specifies `precision`, which is the data format of the register being written. `precision` can be used only for holding registers.

`write(m,'holdingregs',address,values,serverId,'precision')` additionally specifies `serverId` and data format `precision`. You can both arguments together when the write target is holding registers.

Examples

Write Coils Over Modbus

If the write target is coils, the function writes a contiguous sequence of 1–1968 coils to either on or off in a remote device. A coil is a single output bit. A value of `1` indicates the coil is on and a value of `0` means it is off.

Write to 4 coils, starting at address 8289. The `address` parameter is the starting address of the coils to write to, and it is a double. The `values` parameter is an array of values to write.

```
write(m,'coils',8289,[1 1 0 1])
```

You can also create a variable for the values to write.

```
values = [1 1 0 1];
write(m,'coils',8289,values)
```

Write Holding Registers Over Modbus

If the write target is holding registers, the function writes a block of 1–123 contiguous registers in a remote device. Values whose representation is greater than 16 bits are stored in consecutive register addresses.

Set the register at address 49153 to 2000.

```
write(m,'holdingregs',49153,2000)
```

Specify Server ID and Precision Options for the Write Operation

You can write to coils or holding registers and also specify the optional parameter for server ID, and you can specify precision for holding registers. You can set either option by itself or set both the `serverId` option and the `precision` option together. Both options should be listed after the required arguments.

Write 3 values, starting at address 29473, at Server ID 2, converting to `single` precision.

```
write(m,'holdingregs',29473,[928.1 50.3 24.4],2,'single')
```

Input Arguments

`target` — Target area to write to
character vector | string

Target area to write to, specified as a character vector or string. You can perform a Modbus write operation on two types of targets: coils and holding registers, so you must set the target type as either `'coils'` or `'holdingregs'`. Target must be the first argument after the object name. This example writes to 4 coils starting at address 8289.

Example: `write(m,'coils',8289,[1 1 0 1])`

Data Types: `char`

`address` — Starting address to write to
double

Starting address to write to, specified as a double. Address must be the second argument after the object name. This example writes to 6 coils starting at address 5200.

Example: `write(m,'coils',5200,[1 1 0 1 1 0])`

Data Types: `double`

`values` — Array of values to write
double | array of doubles

Array of values to write, specified as a double or array of doubles. `values` must be the third argument after the object name. If the target is coils, valid values are `0` and `1`. If the target is holding registers, valid values must be in the range of the specified precision. You can include the array of values in the syntax, as shown here, or use a variable for the values.

This example writes to 4 coils starting at address 8289.

Example: `write(m,'coils',8289,[0 1 0 1])`

Data Types: `double`

`serverId` — Address of the server to send the write command to
double

Address of the server to send the write command to, specified as a double. Server ID must be specified after the object name, target, address, and values. If you do not specify a `serverId`, the default of `1` is used. Valid values are `0-247`, with `0` being the broadcast address. This example writes 8 coils starting at address 1 from server ID 3.

Example: `write(m,'coils',1,[1 1 1 1 0 0 0 0],3);`

Data Types: `double`

`precision` — Data format of the register being written to on the Modbus server
character vector | string

Data format of the register being written to on the Modbus server, specified as a character vector or string. Precision must be specified after the object name, target, address, and values. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

Note that `precision` does not refer to the return type, which is always `'double'`. It specifies how to interpret the register data.

This example writes to 4 holding registers starting at address 2 using a precision of `'uint32'`.

Example: `write(m,'holdingregs',2,[100 200 300 500],'uint32');`

Data Types: `char`

Version History
Introduced in R2017a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`modbus` | `read` | `writeRead` | `maskWrite`

Topics
"Create a Modbus Connection" on page 18-3
"Configure Properties for Modbus Communication" on page 18-5
"Write Data to a Modbus Server" on page 18-14

writeasync

Asynchronously write values to group or items

Syntax

```
TransID = writeasync(GObj,Values)
TransID = writeasync(IObj,Values)
```

Description

`TransID = writeasync(GObj,Values)` asynchronously writes values to all the items contained in the `dagroup` object `GObj`. `Values` is a cell array of values and is the same size as the number of items in `GObj`. `TransID` is a unique transaction ID for the asynchronous request.

`TransID = writeasync(IObj,Values)` asynchronously writes values to all the items contained in the vector of `daitem` objects specified by `IObj`.

To ensure that a specific value is written to the correct item object, you should construct the `Values` cell array based on the order of the items returned by the `Item` property. Because the values are written to the device, this operation might be time consuming.

The data types of the values do not need to match the canonical data type of the associated items. If a data type conversion cannot be performed, a warning is issued.

When the asynchronous write operation completes, a write async event is generated by the server. If a callback function file is specified for the `WriteAsyncFcn` property, then the function executes when the event is generated.

Note The behavior of an OPC server when writing `NaN` to an item is server-dependent. If you attempt to write `NaN` to an OPC server, the value might be silently ignored by the OPC server. That is, the server might not generate any events in response to writing `NaN` to an item.

Examples

Configure a client, group, and items, for the Matrikon Simulation Server:

```
da = opcda('localhost', 'Matrikon.OPC.Simulation');
connect(da);
grp = addgroup(da, 'ExWrite');
itm = additem(grp, {'Bucket Brigade.Real8', ...
    'Bucket Brigade.String'});
```

Configure the `WriteAsyncFcn` callback to read from the group:

```
grp.WriteAsyncFcn = 'r=read(grp,''device'')';
```

Write values asynchronously to the group:

```
writeasync(grp, {123.456, 'MATLAB is great!'})
```

Version History

Introduced before R2006a

See Also

`cancelasync` | `read` | `readasync` | `refresh` | `write`

writeRead

Perform write and read operation on groups of holding registers in single Modbus transaction

Syntax

```
moddata = writeRead(m,writeAddress,values,readAddress,readCount)
moddata = writeRead(m,writeAddress,values,writePrecision,readAddress,
readCount,readPrecision)
moddata = writeRead( ___ ,serverId)
```

Description

`moddata = writeRead(m,writeAddress,values,readAddress,readCount)` writes data to Modbus object `m` at the starting address `writeAddress` using the values to write `values`, and then reads data at the starting address `readAddress` using the number of values to read `readCount`.

This function performs a combination of one write operation and one read operation on groups of holding registers in a single Modbus transaction. The write operation is always performed before the read. The range of addresses to read must be contiguous, and the range of addresses to write must be contiguous, but write and read addresses are specified independently and need not overlap.

`moddata = writeRead(m,writeAddress,values,writePrecision,readAddress,readCount,readPrecision)` adds optional precisions for the write and read operations. The `writePrecision` and `readPrecision` arguments specify the data format of the register being written to and read from on the Modbus server.

`moddata = writeRead( ___ ,serverId)` additionally uses the `serverId` as the address of the server to send the command to.

Examples

Write and Read Holding Registers

The `writeRead` function is used to perform a combination of one write operation and one read operation on groups of holding registers in a single Modbus transaction. The write operation is always performed before the read. The range of addresses to read must be contiguous, and the range of addresses to write must be contiguous, but each is specified independently and may or may not overlap.

Write 2 holding registers starting at address 300, and read 4 holding registers starting at address 17250.

```
moddata = writeRead(m,300,[500 1000],17250,4)
```

```
moddata =

   35647   48923   50873   60892
```

If the operation is successful, it returns an array of doubles, each representing a 16-bit register value, where the first value in the vector corresponds to the register value at the address specified in `readAddress`.

You can optionally create variables for the values to be written, instead of including the array of values in the function syntax. The example could be written this way, using a variable for the values:

```
values = [500 1000];
moddata = writeRead(m,300,values,17250,4)
```

```
moddata =

   35647   58923   50873   60892
```

Write and Read Holding Registers, and Specify Server ID

Use the `serverId` argument to specify the address of the server to send the command to.

Write 3 holding registers starting at address 400, and read 4 holding registers starting at address 52008 from server ID 6.

```
moddata = writeRead(m,400,[1024 512 680],52008,4,6)
```

```
moddata =

   38629   24735   29456   39470
```

Write and Read Holding Registers, and Specify Precisions

Use the `writePrecision` and `readPrecision` arguments to specify the data format of the register being read from or written to on the Modbus server.

Write 3 holding registers starting at address 500, and read 6 holding registers starting at address 52008 from server ID 6. Specify a `writePrecision` of `'uint64'` and a `readPrecision` of `'uint32'`.

```
moddata = writeRead(m,500,[1024 512 680],'uint64',52008,6,'uint32',6)
```

```
moddata =

   38629   24735   29456   39470   33434   29484
```

Input Arguments

`writeAddress` — Starting address of the registers to write

double

Starting address to write to, specified as a double. `writeAddress` must be the first argument after the object name. This example writes 2 holding registers starting at address 501 and reads 4 holding registers starting at address 11250. The `writeAddress` is `501`.

Example: `writeRead(m,501,[1024 512],11250,4)`

Data Types: `double`

`values` — Array of values to write
double | array of doubles

Array of values to write, specified as a double or array of doubles. Values must be the second argument after the object name. Each value must be in the range `0`–`65535`. This example writes 2 holding registers starting at address 501 and reads 4 holding registers starting at address 11250. The `values` are `[1024 512]`.

Example: `writeRead(m,501,[1024 512],11250,4)`

Data Types: `double`

`readAddress` — Starting address of the holding registers to read
double

Starting address of the holding registers to read, specified as a double. `readAddress` must be the third argument after the object name. This example writes 2 holding registers starting at address 501 and reads 4 holding registers starting at address 11250. The `readAddress` is `11250`.

Example: `writeRead(m,501,[1024 512],11250,4)`

Data Types: `double`

`readCount` — Number of holding registers to read
double

Number of holding registers to read, specified as a double. `readCount` must be the fourth argument after the object name. This example writes 2 holding registers starting at address 501 and reads 4 holding registers starting at address 11250. The `readCount` is `4`.

Example: `writeRead(m,501,[1024 512],11250,4)`

Data Types: `double`

`serverId` — Address of the server to send the command to
double

Address of the server to send the command to, specified as a double. Server ID must be specified after the object name, write address, values, read address, and read count. If you do not specify a `serverId`, the default of `1` is used. Valid values are `0`–`247`, with `0` being the broadcast address. This example writes 3 holding registers starting at address 400 and reads 4 holding registers starting at address 52008 from server ID 6.

Example: `writeRead(m,400,[1024 512 680],52008,4,6)`

Data Types: `double`

`writePrecision` — Data format of the holding register being written to on the Modbus server
character vector | string

Data format of the holding register being written to on the Modbus server, specified as a character vector or string. `writePrecision` must be specified after the write address and values. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

Note that `writePrecision` does not refer to the return type, which is always `'double'`. It specifies how to interpret the register data.

This example writes 3 holding registers starting at address 400 and reads 4 holding registers starting at address 52008 from server ID 6. It also specifies a `writePrecision` of `'uint64'`.

Example: `writeRead(m,400,[1024 512 680],'uint64',52008,4,'uint32',6)`

Data Types: `char`

`readPrecision` — Data format of the holding register being read from on the Modbus server
character vector | string

Data format of the holding register being read from on the Modbus server, specified as a character vector or string. `readPrecision` must be specified after the read address, and read count. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

Note that `readPrecision` does not refer to the return type, which is always `'double'`. It specifies how to interpret the register data.

This example writes 3 holding registers starting at address 400 and reads 4 holding registers starting at address 52008 from server ID 6. It also specifies a `readPrecision` of `'uint32'`.

Example: `writeRead(m,400,[1024 512 680],'uint64',52008,4,'uint32',6)`

Data Types: `char`

Output Arguments

`moddata` — Value of read data
double

Read data values, returned as a double or array of doubles.

Version History

Introduced in R2017a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`modbus` | `read` | `write` | `maskWrite`

Topics
“Create a Modbus Connection” on page 18-3
“Configure Properties for Modbus Communication” on page 18-5
“Write and Read Multiple Holding Registers” on page 18-16

writeValue

Package: `opc.ua`

Write values to nodes on OPC UA server

Syntax

```
writeValue(UaClient,NodeList,Values)
writeValue(NodeList,Values)
```

Description

`writeValue(UaClient,NodeList,Values)` writes content of `Values`, to the nodes identified by `NodeList`. You can browse for node objects using `browseNamespace`. You can also create nodes using `opcuanode`.

If `NodeList` is a single node, then `Values` is the value written to the node. If `NodeList` is an array of nodes, `Values` must be a cell array the same size as `NodeList`, and each element of the cell array is written to the corresponding element of `NodeList`.

The data type of the value you are writing does not need to match the node `ServerDataType` property. All values are automatically converted before writing to the server. However, a warning or error is generated if the data type conversion fails. For `DateTime` data types, you can pass a MATLAB datetime or a number; any numeric value can be interpreted as a MATLAB datetime.

To confirm what size arrays can be written to a node, check the `ServerValueRank` and `ServerArrayDimensions` properties of the node:

- A `ServerValueRank` value of `-3` indicates a scalar or 1-dimensional array, `-2` indicates any size array, `-1` indicates a scalar, `0` indicates an array with 1 or more dimensions, and a positive value indicates the number of dimensions.
- If the number of dimensions is fixed, `ServerArrayDimensions` is an array specifying the maximum possible length of each dimension. A value of `0` for a dimension length indicates no limit.

  For example, if a node supports 2-dimensional arrays of a maximum size of 64-by-32, `ServerValueRank` has a value of `2` and `ServerArrayDimensions` `[64, 32]`.

`writeValue(NodeList,Values)` writes content of `Values`, to the nodes identified by `NodeList`. All nodes must be of the same connected client.

Examples

Write a Value to a Node

Write a new value to the Static Double node on a local server.

```
uaClient = opcua('localhost', 53530);
connect(uaClient);
```

```
staticNode = findNodeByName(uaClient.Namespace, 'StaticData', '-once');
scalarNode = findNodeByName(staticNode, 'StaticVariables', '-once');
dblNode = findNodeByName(staticNode, 'Double');
writeValue(uaClient, dblNode, 3.14159)
[newVal,newTS] = readValue(uaClient, dblNode)
```

Write multiple values to a single node.

```
arrayNode = opcuanode(6, 'DoubleArray', uaClient);
writeValue(arrayNode, [3.14, 1.212]);
```

Write scalar values to multiple nodes.

```
multiNodes = opcuanode(6, {'Double','Float'}, uaClient);
writeValue(multiNodes, {34,12});
```

Input Arguments

UaClient — OPC UA client
OPC UA client object

OPC UA client specified as an OPC UA client object. The client must be connected.

NodeList — List of nodes
array of node objects

List of nodes specified as an array of node objects or a single node. For information on node object functions and properties, type:

```
help opc.ua.Node
```

Values — values
cell array | scalar | array

Values specified as a scalar, array, or cell array values. If writing to a single node, use a scalar or array of values. If writing to an array of nodes, use a cell array of values; each element of the call array is written to the corresponding node.

Version History
Introduced in R2015b

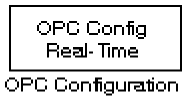
See Also

Functions
`getNamespace` | `browseNamespace` | `readValue`

Blocks

OPC Configuration

Configure OPC DA clients for model, pseudo real-time control, and behavior for OPC errors and events



Libraries:
Industrial Communication Toolbox

Description

The OPC Configuration block defines the OPC Data Access clients to be used in a model, configures pseudo real-time behavior for the model, and defines behavior for OPC errors and events.

The block has no input ports. One optional output port displays model latency.

You cannot place more than one OPC Configuration block in a model. If you attempt to do so, an error message appears, and the second OPC Configuration block is disabled.

Ports

Output

Pseudo real-time latency — Wait time for each simulation step
vector of double

Outputs the model latency as the time spent waiting at each simulation step to achieve pseudo real-time behavior.

Data Types: `double`

Parameters

OPC Configuration

Configure OPC Clients — Define and configure OPC DA clients for use throughout the model

Click to open the OPC Client Manager dialog for the model. Each model has a list of clients associated with it. These clients are used during the simulation to read or write data to an OPC DA server. For more information, see “Use the OPC Client Manager” on page 10-11.

Error Control

These parameters define actions when OPC-specific errors and events are encountered. The available actions are to produce an error and stop the simulation, produce a warning and continue the simulation, or ignore the error or event.

Items not available on server — Behavior on missing items
`Error` (default) | `Warn` | `None`

Defines the behavior for items that are specified in a Read or Write block but do not exist on the server when the simulation starts.

Read/write errors — Behavior on read or write errors
`Warn` (default) | `Error` | `None`

Defines the behavior when a read or write operation fails.

Server unavailable — Behavior on server shutdown
`Error` (default) | `Warn` | `None`

Defines the behavior when the client cannot connect to the OPC DA server, or when the server sends a shutdown event to the client.

Pseudo real-time violation — Behavior on real-time violation
`Warn` (default) | `Error` | `None`

Defines the behavior when the simulation runs slower than real time. See the **Pseudo real-time simulation** options for more information.

Pseudo real-time simulation

Enable pseudo real-time simulation — Control simulation speed
on (default) | off

This parameter allows you to configure options for running the simulation in pseudo real time. When checked (on), the model execution time matches the system clock as closely as possible by slowing down the simulation appropriately. Note that the real-time control settings do not guarantee real-time behavior.

If the model runs slower than real time, a pseudo real-time latency violation error occurs. You can control how Simulink responds to a pseudo real-time latency violation using the settings in the **Error control** pane.

Speedup — Simulation speedup factor
1 (default) | integer value

The **Speedup** setting determines how many times faster than the system clock the simulation runs. For example, a setting of 2 means that a 10-second simulation will take 5 seconds to complete. The **Speedup** parameter must be a literal integer; you cannot use a MATLAB or Simulink model workspace variable to define the speedup factor.

Output Ports

Show pseudo real-time latency port — Add latency output port
off (default) | on

Check this parameter (on) to add an output port to the block for the model pseudo real-time latency.

Version History

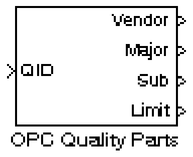
Introduced before R2006a

See Also

Blocks
OPC Read | OPC Write

OPC Quality Parts

Convert OPC DA quality ID into vendor, major, minor, and limit status



Libraries:
Industrial Communication Toolbox

Description

The OPC Quality Parts block converts an OPC Data Access quality ID vector into four parts:

- Vendor status
- Major quality
- Quality substatus
- Limit status

The Quality output port of an OPC Read block generates quality IDs. For more information on quality parts, see "OPC Quality" on page A-2.

Ports

Input

QID — Quality ID
OPC quality ID integer

OPC quality ID integer, typically connected to the Quality output port of an OPC Read block.

Data Types: `uint16`

Output

Vendor — Vendor quality information
integer 0-255

Quality ID information specific to the vendor.

Data Types: `uint16`

Major — Major quality value
0 | 1 | 3

Major quality value, indicating bad (0), uncertain (1), or good (3) quality. For more information, see "Major Quality" on page A-3.

Data Types: `uint16`

Sub — Quality substatus value
integer 0-7

Each major quality status has an additional substatus that describes the quality of the value in more detail. The interpretation of the substatus value depends on whether its major quality is good, uncertain, or bad. See “Quality Substatus” on page A-4.

Data Types: `uint16`

Limit — Limit status
0 | 1 | 2 | 3

The limit status of the quality value, indicating that the value is not limited (0), fixed at a lower limit (1), fixed at an upper limit (2), or constant (3). See “Limit Status” on page A-6.

Data Types: `uint16`

Version History

Introduced before R2006a

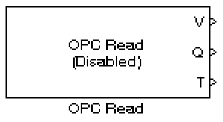
See Also

Blocks
OPC Read

Topics
“OPC Quality” on page A-2

OPC Read

Read data from OPC DA server



Libraries:
Industrial Communication Toolbox

Description

The OPC Read block reads data from one or more items on an OPC Data Access server. The read operation takes place synchronously (from the cache or from the device) or asynchronously (from the device).

The block outputs the values (`V`) of the requested items in the first output, and optionally outputs the quality IDs (`Q`) and the time stamps (`T`) associated with each data value in separate outputs. The time stamp can be output as a serial date number (real-world time), or as the number of seconds from the start of the simulation (simulation time).

The V,Q,T triple available at the output ports is the last known data for each of the items read by the block. Use the time stamp output to determine when a sample last changed.

Note You must have an OPC Configuration block in your model to use the OPC Read block. You cannot open the OPC Read parameters dialog without first including an OPC Configuration block in the model.

Ports

Output

V — Values of requested items
item value data

Values of the requested items, returned as a vector of type specified by the **Value port data type** parameter.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Q — Quality of requested items
vector of `uint16`

Quality of requested items, returned as a vector of `uint16`. For details on the quality format, see “OPC Quality” on page A-2. This port is optional, depending on the setting of the **Show quality port** parameter.

Data Types: `uint16`

T — Time stamps of requested items
seconds | date number

The time stamp can be output as a vector of serial date number (real-world time), or as the number of seconds from the start of the simulation (simulation time). This port is optional, depending on the setting of the **Show timestamp port as** parameter.

Data Types: `double`

Parameters

Import from Workspace — Import block settings from `dagroup` object in workspace

The import dialog allows you to import settings for the OPC Read block from a `dagroup` object in the MATLAB base workspace. The client, item IDs, and sample time are updated based on the properties of the imported group. The **Value port data type** is also set if all items in the group have the same `DataType` property.

Client — Define OPC client for block
list choice

Defines the OPC DA client associated with this block. You can add clients to the list using **Configure OPC Clients**. For more information, see “Use the OPC Client Manager” on page 10-11.

Item IDs — List of OPC server items
list view

Shows the items to be read from the specified server. You can add items to the list using **Add Items**, or delete items using **Delete**. You can change the order of the items in the list using **Move Up** or **Move Down**. The order of the items determines the order of their values in the block outputs.

Read mode — Set synchronous reading
`Synchronous (cache)` (default) | `Synchronous (device)` | `Asynchronous`

Defines the read mode for this block. Available options are `Asynchronous`, `Synchronous (cache)`, or `Synchronous (device)`. Synchronous reads are generally more reliable, but have slightly more overhead than asynchronous reads.

Sample time — Sample time for block reads
0.5 (default) | numeric

Defines the sample time for the block, in seconds. For synchronous reads, data is read from the server at the specified sample time. For asynchronous reads, the sample time setting defines the update rate for data change events.

Value port data type — Data type for value
`double` (default) | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `logical`

Defines the data type for the value output port. The OPC DA server is responsible for converting all data to the required type. See DataType.

Note For items with a CanonicalDataType of `logical` on the server, you can set the OPC Read block to output a numeric value type:

- When the item value is `true`, the OPC Read block value output is `-1` for signed data types, or the maximum value for unsigned integers.

- When the item value is `false`, the block value output is `0`.

Show quality port — Add quality output to block
on (default) | off

When checked (on), the quality IDs of all the items are output in the second port as a vector of unsigned 16-bit integers (`uint16`). Use the OPC Quality Parts block to separate the quality ID into component parts.

Show timestamp port as — Set time stamp basis
on (default) | off

When checked (on), the timestamps for each of the items are output in the last port as a vector of doubles. You can choose whether to output the timestamps as **Seconds since start** (i.e., simulation time) or as **Serial date numbers** (i.e., real-world time).

Version History

Introduced before R2006a

See Also

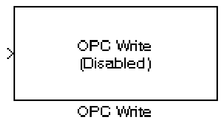
Blocks
OPC Configuration | OPC Quality Parts | OPC Write

Functions
`genslread`

Topics
"OPC Quality" on page A-2

OPC Write

Write data to OPC DA server



Libraries:
Industrial Communication Toolbox

Description

The OPC Write block writes data to one or more items on an OPC Data Access server. You can choose the write operation to take place synchronously or asynchronously.

Each element of the input vector is written to the corresponding item in the item ID list defined for the OPC Write block.

Note You must have an OPC Configuration block in your model to use the OPC Write block. You cannot open the OPC Write dialog without first including an OPC Configuration block in the model.

Ports

Input

Value — Vector of item values
vector

Values to be written to OPC items. Each element of the input vector is written to a separate item on the OPC DA server.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `Boolean`

Parameters

Import from Workspace — Import block settings from `dagroup` object in workspace

The import dialog allows you to import settings for the OPC Write block from a `dagroup` object in the MATLAB base workspace. The client, item IDs, and sample time are updated based on the properties of the imported group.

Client — Define OPC client for block
list choice

Defines the OPC DA client associated with this block. You can add clients to the list using **Configure OPC Clients**. For more information, see “Use the OPC Client Manager” on page 10-11.

Item IDs — List of OPC server items
list view

Shows the items to be written to on the specified server. You can add items to the list using **Add Items**, or delete items using **Delete**. You can change the order of the items in the list using **Move Up** or **Move Down**. Each element of the vector at input port is written in order to the corresponding item in the list.

Write mode — Set synchronous writing
`Synchronous` (default) | `Asynchronous`

Defines the write mode for this block, as `Synchronous` or `Asynchronous`. Synchronous writes are generally more reliable than asynchronous, but have slightly more overhead.

Sample time — Block sample time
0 (default) | numeric

Defines the sample time for the block. Data is written to the server at the specified sample time. You can specify `0` for continuous mode, or `-1` to inherit the sample time of the block connected to the input of the OPC Write block.

Version History

Introduced before R2006a

See Also

Blocks
OPC Configuration | OPC Read

Functions
`genslwrite`

Industrial Communication Toolbox Examples

- “Install a Simulation Server for OPC Examples” on page 21-2
- “Acquire Data from an OPC Data Access Server” on page 21-3
- “Locate and Browse OPC Data Access Servers” on page 21-7
- “Create and Configure OPC Objects” on page 21-10
- “Manage OPC Data Access Objects” on page 21-14
- “Read and Write Data to an OPC Data Access Server” on page 21-17
- “Log Data from an OPC Data Access Server” on page 21-21
- “View the OPC Event Log” on page 21-24
- “Monitor Logging Progress with Callbacks” on page 21-26
- “Update MATLAB Plots While Logging OPC Data” on page 21-28
- “Locate and Browse OPC Historical Data Access Servers” on page 21-30
- “Acquire Data from an OPC Historical Data Access Server” on page 21-33
- “Visualize and Preprocess OPC HDA Data” on page 21-37
- “Browse OPC UA Server Namespace” on page 21-44
- “Read and Write Current OPC UA Server Data” on page 21-51
- “Read Historical OPC UA Server Data” on page 21-56
- “Visualize and Preprocess OPC UA Data” on page 21-61
- “Read and Write to an OPC Data Access Server from Simulink” on page 21-70
- “Use OPC Data to Test a Binary Distillation Column Plant Model” on page 21-72
- “Get Started Accessing Data from a PI Server” on page 21-74
- “Read Data from a PI Server” on page 21-77
- “Process PI Data Using Common MATLAB Operations” on page 21-81
- “Get Started with MQTT” on page 21-88
- “Get Data from Subscribed Topics in an MQTT Client” on page 21-90
- “Subscribe to an MQTT Topic with a Callback Function” on page 21-93
- “Subscribe to an MQTT Wildcard Topic” on page 21-95

Install a Simulation Server for OPC Examples

This example shows you how to install a simulated OPC Server for use with the OPC examples.

Many of the OPC examples need to connect to a live OPC server. Matrikon™, a supplier of a variety of OPC servers, provides a simulation server for testing purposes. This example explains how to download and install that simulation server, and test that MATLAB® can connect to the server.

Note: You must have administrator privileges on your machine in order to install the Matrikon OPC Simulation Server correctly.

Download the Matrikon OPC Simulation Server

Download the simulation server by visiting https://www.matrikonopc.com/ and downloading the "OPC Simulation Server".

You may be required to register with Matrikon in order to download the OPC Simulation Server.

Install the OPC Simulation Server

Perform a default installation of the Matrikon OPC Simulation Server, including all prerequisites.

Run OPCREGISTER (64-bit users only)

If you are running 64-bit MATLAB, you should re-register the OPC Foundation Core Components that ship with MATLAB. This enables the 64-bit MATLAB application to browse for 32-bit servers on your machine.

```
opcregister('-silent')
```

Verify the Existence of the OPC Simulation Server

Browse for OPC servers on your local machine to verify that the OPC Simulation Server has been successfully installed.

```
sInfo = opcserverinfo('localhost')
```

```
sInfo = 
                 Host: 'localhost'
             ServerID: {'Matrikon.OPC.Simulation.1'  'OSI.DA.1'  'OSI.HDA.1'}
    ServerDescription: {1x3 cell}
     OPCSpecification: {'DA2'  'DA2'  'DA2'}
    ObjectConstructor: {1x3 cell}
```

The list of ServerIDs should include `Matrikon.OPC.Simulation.1`

Acquire Data from an OPC Data Access Server

This example shows how to use Industrial Communication Toolbox™ to acquire data from an OPC server.

PREREQUISITES:

- “Install a Simulation Server for OPC Examples” on page 21-2

Create OPC Data Access Object Hierarchy

Create an `opcda` object associated with the required server and connect to the server.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1')
connect(da)
```

```
da =

Summary of OPC Data Access Client Object: localhost/Matrikon.OPC.Simulation.1

   Server Parameters
      Host      : localhost
      ServerID  : Matrikon.OPC.Simulation.1
      Status    : disconnected
      Timeout   : 10 seconds

   Object Parameters
      Group     : 0-by-1 dagroup object
      Event Log : 0 of 1000 events
```

Create a group object to manage the required items.

```
grp = addgroup(da,'DemoGroup')
```

```
grp =

Summary of OPC Data Access Group Object: DemoGroup

   Object Parameters
      Group Type   : private
      Item         : 0-by-1 daitem object
      Parent       : localhost/Matrikon.OPC.Simulation.1
      Update Rate  : 0.5
      Deadband     : 0%

   Object Status
      Active       : on
      Subscription : on
      Logging      : off

   Logging Parameters
      Records      : 120
      Duration     : at least 60 seconds
      Logging to   : memory
      Status       : Waiting for START.
                     0 records available for GETDATA/PEEKDATA
```

Add the `Real8` item from `Saw-Toothed Waves` and the `Real8` and `UInt2` items from `Triangle Waves` to the group.

```
itmIDs = {'Saw-toothed Waves.Real8', ...
    'Triangle Waves.Real8', ...
    'Triangle Waves.UInt2'};
itm = additem(grp,itmIDs)
```

```
itm =

   OPC Item Object Array:

   Index:  Active:  ItemID:             Value:                Quality:    TimeStamp:
   1       on       ...hed Waves.Real8                        Bad: Ou...
   2       on       ...gle Waves.Real8                        Bad: Ou...
   3       on       ...gle Waves.UInt2                        Bad: Ou...
```

Configure OPC Object Properties

Configure the group to log 60 seconds of data at 0.2 second intervals.

```
logDuration = 60;
logRate = 0.2;
numRecords = ceil(logDuration./logRate)
grp.UpdateRate = logRate;
grp.RecordsToAcquire = numRecords;
```

```
numRecords =

   300
```

Acquire OPC Server Data

Start the acquisition task, and wait for the task to complete before continuing execution of any MATLAB™ code.

```
start(grp)
wait(grp)
```

Note that while waiting for a logging task to complete, MATLAB continues to process callbacks from OPC objects (and other objects that include callback functionality).

Retrieve the logged data into separate arrays for the time stamps, quality, and values.

```
[logIDs,logVal,logQual,logTime,logEvtTime] = getdata(grp,'double');
```

Examine the workspace for the sizes of the data.

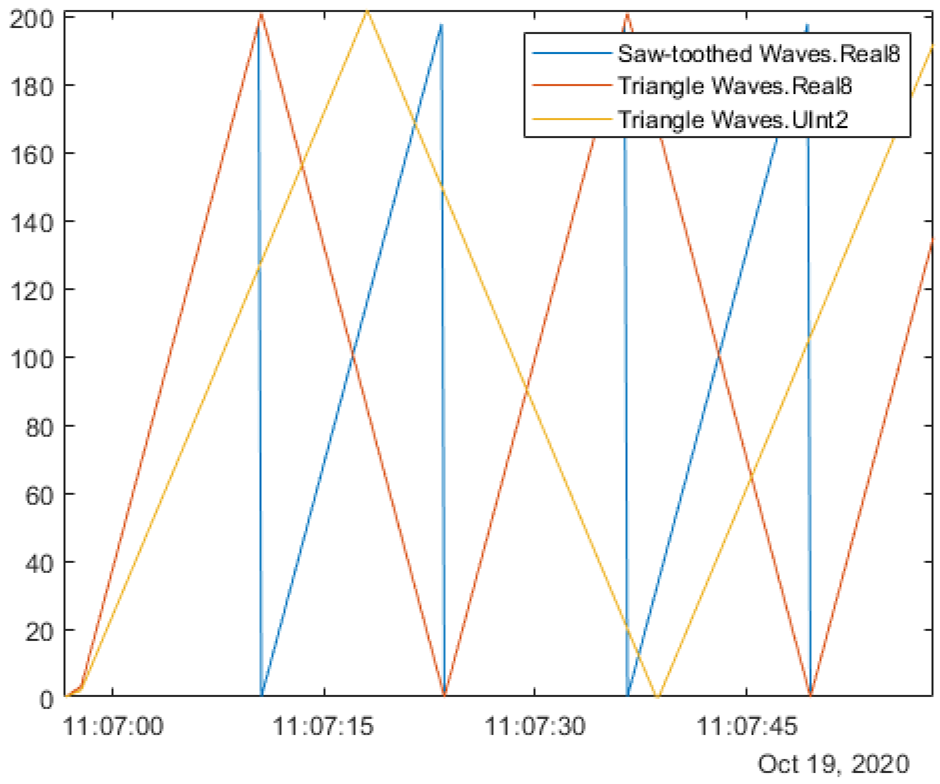
```
whos log*
```

```
  Name                  Size                Bytes  Class      Attributes
  logDuration       1x1                  8  double
  logEvtTime      300x1               2400  double
  logIDs            1x3                438  cell
  logQual         300x3             126004  cell
  logRate           1x1                  8  double
  logTime         300x3               7200  double
  logVal          300x3               7200  double
```

Plot the Data

You can now plot this data all on one set of axes.

```
logTime = datetime(logTime,'ConvertFrom','datenum');
plot(logTime,logVal);
axis tight
lgd = legend(logIDs);
lgd.AutoUpdate = 'off';
```

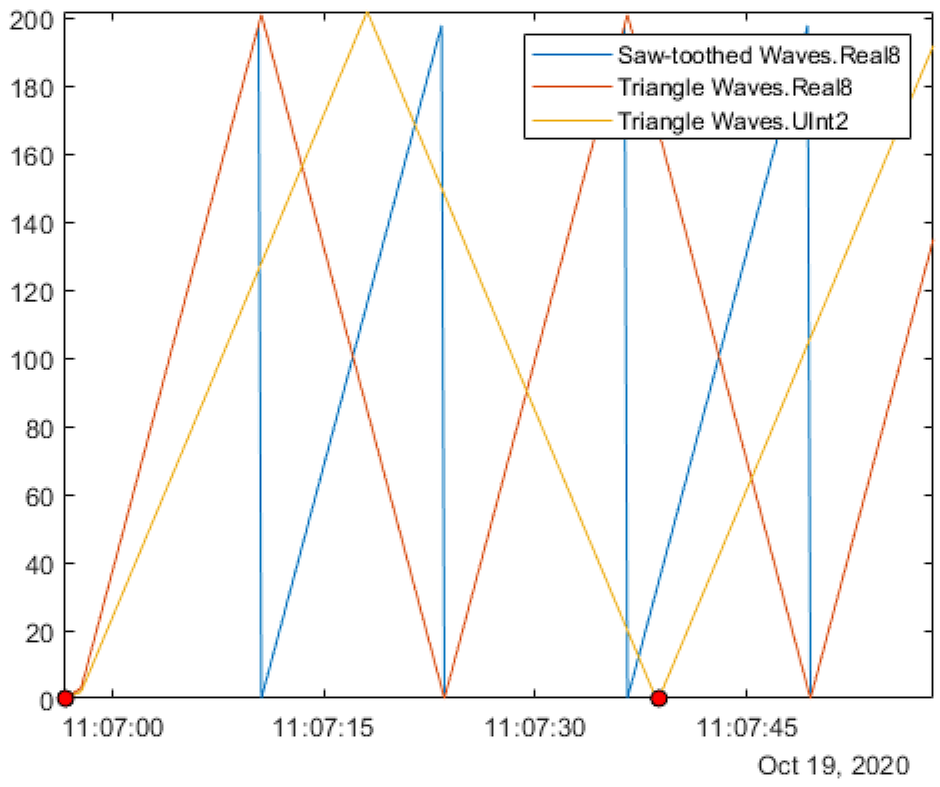


The value data does not provide the full picture. You should always examine the quality of the data to determine the validity of the value array.

Annotate the plot with markers where the quality is not `Good`.

```
hold on
isBadQual = strncmp(logQual,'Bad',3);
isRepeatQual = strncmp(logQual,'Repeat',6);
for k = 1:size(logQual,2)
```

```
    badInd = isBadQual(:,k);
    plot(logTime(badInd,k),logVal(badInd,k),'ro', ...
        'MarkerFaceColor','r','MarkerEdgeColor','k')
    repInd = isRepeatQual(:,k);
    plot(logTime(repInd, k),logVal(repInd,k),'ro', ...
        'MarkerFaceColor',[0.8 0.5 0],'MarkerEdgeColor','k')
end
hold off
```



Bad quality is marked in red, and Repeat quality is marked in orange.

Clean Up

Disconnect and delete the client object from the OPC engine.

```
disconnect(da)
delete(da)
```

Locate and Browse OPC Data Access Servers

This example shows you how to browse the network for OPC servers, and query the server name space for server items and their properties.

PREREQUISITES:

- "Install a Simulation Server for OPC Examples" on page 21-2

Step 1: Browse the Network for OPC Servers

You use the `opcserverinfo` function to query a host on the network for available OPC Data Access servers. This example uses the local host.

```
hostInfo = opcserverinfo('localhost')
```

```
hostInfo = 

                 Host: 'localhost'
             ServerID: {'Matrikon.OPC.Simulation.1'}
    ServerDescription: {'MatrikonOPC Server for Simulation and Testing'}
     OPCSpecification: {'DA2'}
    ObjectConstructor: {'opcda('localhost', 'Matrikon.OPC.Simulation.1')'}
```

The returned structure provides information about each server:

```
hostInfo.ServerDescription'
```

```
ans = 

    'MatrikonOPC Server for Simulation and Testing'
```

and about the Server ID you use to create a client object.

```
allID = hostInfo.ServerID'
```

```
allID = 

    'Matrikon.OPC.Simulation.1'
```

Step 2: Construct a Client Object and Connect to the Server

Use the host name and server ID found in the previous step to construct a client object.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1')
```

```
da = 

Summary of OPC Data Access Client Object: localhost/Matrikon.OPC.Simulation.1
```

```
   Server Parameters
      Host      : localhost
      ServerID  : Matrikon.OPC.Simulation.1
      Status    : disconnected
      Timeout   : 10 seconds

   Object Parameters
      Group     : 0-by-1 dagroup object
      Event Log : 0 of 1000 events
```

Connect the client to the server.

```
connect(da);
```

Step 3: Retrieve the Server Name Space

Retrieve the name space of the server.

```
ns = getnamespace(da)
```

```
ns = 

4×1 struct array with fields:

    Name
    FullyQualifiedID
    NodeType
    Nodes
```

Each element of the structure is a node in the server name space.

```
ns(1)
```

```
ans = 

                Name: 'Simulation Items'
    FullyQualifiedID: 'Simulation Items¥'
            NodeType: 'branch'
               Nodes: [8×1 struct]
```

Step 4: Find Items in the Name Space

Use the `serveritems` function to find all items in the name space containing the string `Real`.

```
realItems = serveritems(ns,'*Real*')
```

```
realItems = 

    'Bucket Brigade.ArrayOfReal8'
    'Bucket Brigade.Real4'
    'Bucket Brigade.Real8'
    'Random.ArrayOfReal8'
    'Random.Real4'
    'Random.Real8'
    'Read Error.ArrayOfReal8'
    'Read Error.Real4'
    'Read Error.Real8'
    'Saw-toothed Waves.Real4'
    'Saw-toothed Waves.Real8'
    'Square Waves.Real4'
    'Square Waves.Real8'
    'Triangle Waves.Real4'
    'Triangle Waves.Real8'
    'Write Error.ArrayOfReal8'
    'Write Error.Real4'
    'Write Error.Real8'
    'Write Only.ArrayOfReal8'
    'Write Only.Real4'
    'Write Only.Real8'
```

Step 5: Query Server Item Properties

Examine the Canonical Data Type (`PropID = 1`) and the Item Access Rights (`PropID = 5`) of the second item found.

```
canDT = serveritemprops(da,realItems{2},1)
accessRights = serveritemprops(da,realItems{2},5)
```

```
canDT = 

             PropID: 1
    PropDescription: 'Item Canonical DataType'
          PropValue: 'single'
         PropItemID: ''


accessRights = 

             PropID: 5
    PropDescription: 'Item Access Rights'
          PropValue: 'read/write'
         PropItemID: ''
```

Step 6: Clean Up OPC Objects

Disconnect the client from the server and remove OPC objects from memory when you no longer need them. Deleting the client object also deletes the group and item objects.

```
disconnect(da)
delete(da)
```

Create and Configure OPC Objects

This example shows you how to create and configure objects in the workspace to access an OPC server.

PREREQUISITES:

- "Install a Simulation Server for OPC Examples" on page 21-2

Create Client Objects

Create a client using the `opcda` function. You need the host name and the server ID for the OPC server associated with this client.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
```

Add Groups to the Client

Use the `addgroup` function to add groups to the client object. The toolbox automatically assigns a name to the group, if you do not specify one.

```
grp1 = addgroup(da);
```

Group objects are used to manage collections of `daitem` objects.

To assign your own name to a group, the name must be unique for all the groups in a client. Pass the name as an additional argument to `addgroup`.

```
grp2 = addgroup(da,'MyGroup');
```

Type the object name to view a summary of the group object.

```
grp1
```

```

grp1 =

Summary of OPC Data Access Group Object: Group0

   Object Parameters
      Group Type   : private
      Item         : 0-by-1 daitem object
      Parent       : localhost/Matrikon.OPC.Simulation.1
      Update Rate  : 0.5
      Deadband     : 0%

   Object Status
      Active       : on
      Subscription : on
      Logging      : off

   Logging Parameters
      Records      : 120
      Duration     : at least 60 seconds
      Logging to   : memory
      Status       : Waiting for START.
                       0 records available for GETDATA/PEEKDATA
```

Add Item Objects to the Group

Add the item named `Random.Real8` to the group.

```
itm1 = additem(grp1,'Random.Real8');
```

If you want the value stored in MATLAB® to have a specific data type, specify it as the third argument.

```
itm2 = additem(grp1,'Random.UInt2','double');
```

To view a summary of the object, type the name of the object.

```
itm1
```

```
itm1 =

Summary of OPC Data Access Item Object: Random.Real8

   Object Parameters
      Parent        : Group0
      Access Rights : read

   Object Status
      Active        : on

   Data Parameters
      Data Type     : double
      Value         : 0
      Quality       : Bad: Out of Service
      Timestamp     : 12-Apr-2016 16:19:50
```

Create Object Vectors

References to multiple OPC objects can be stored in object vectors.

```
itmVec = [itm1,itm2]
```

```
itmVec =

   OPC Item Object Array:

   Index:  Active:  ItemID:             Value:                Quality:    TimeStamp:
   1       on       Random.Real8        0                     Bad: Ou...  16:19:50
   2       on       Random.UInt2                              Bad: Ou...
```

Displaying the object vector shows information about each object in the vector.

View and Change Object Properties

View a list of all properties supported by the object.

```
get(da)
```

```
  General Settings:
    EventLog = []
    EventLogMax = 1000
    Group = [1×2 dagroup]
    Host = localhost
    Name = localhost/Matrikon.OPC.Simulation.1
    ServerID = Matrikon.OPC.Simulation.1
    Status = connected
    Tag = 
    Timeout = 10
    Type = opcda
    UserData = []

  Callback Function Settings:
    ErrorFcn = @opccallback
    ShutdownFcn = @opccallback
    TimerFcn = []
    TimerPeriod = 10
```

Obtain information about a specific property.

```
clientName = da.Name
```

```
clientName =

localhost/Matrikon.OPC.Simulation.1
```

Get information about a property using the `propinfo` function.

```
statusInfo = propinfo(da,'Status')
```

```
statusInfo = 

               Type: 'string'
         Constraint: 'enum'
    ConstraintValue: {'disconnected'  'connected'}
       DefaultValue: 'disconnected'
           ReadOnly: 'always'
```

The information includes whether the property is read-only, and lists the valid values for properties that have a predefined set of values.

Set the value of the `Timeout` property to 30 seconds.

```
da.Timeout = 30
```

```
da = 

Summary of OPC Data Access Client Object: localhost/Matrikon.OPC.Simulation.1

   Server Parameters
```

```
      Host      : localhost
      ServerID  : Matrikon.OPC.Simulation.1
      Status    : connected
      Timeout   : 30 seconds

   Object Parameters
      Group     : 2-by-1 dagroup object
      Event Log : 0 of 1000 events
```

Clean Up

Delete objects that you are finished using from the OPC engine.

```
disconnect(da)
delete(da)
```

Deleting the client object also deletes the group and item objects associated with that client.

Manage OPC Data Access Objects

This example shows you how to find, create, and remove OPC object in the workspace.

Find OPC Objects in Memory

Use the `opcfind` function to find OPC objects in memory.

```
opcfind
```

```
ans =
     []
```

Create OPC Objects

Create some OPC objects.

```
da = opcda('localhost', 'Dummy.Server.1');
grp = addgroup(da);
itm1 = additem(grp, 'Fake.Item.ID1');
itm2 = additem(grp, 'Fake.Item.ID2');
```

Find all valid objects.

```
allOPC = opcfind
```

```
allOPC =
    [1x1 opcda]    [1x1 dagroup]    [1x1 daitem]    [1x1 daitem]
```

The information is returned in a cell array, because `opcfind` can locate different objects. Use cell indexing to access an object.

```
foundGrp = allOPC{2}
```

```
foundGrp =
Summary of OPC Data Access Group Object: group1
   Object Parameters
      Group Type   : private
      Item         : 2-by-1 daitem object
      Parent       : localhost/Dummy.Server.1
      Update Rate  : 0.5
      Deadband     : 0%
   Object Status
      Active       : on
      Subscription : on
      Logging      : off
   Logging Parameters
      Records      : 120
      Duration     : at least 60 seconds
      Logging to   : memory
      Status       : Waiting for START.
                     0 records available for GETDATA/PEEKDATA
```

Pass property/value pairs to the `opcfind` function to find objects with a specific property.

```
allDA = opcfind('Type', 'opcda')
allDA =
    [1x1 opcda]
```

Remove Objects From Memory

To delete an OPC object from memory, use the `delete` function with the object. Deleting a client object deletes all group and item objects associated with the client. Deleting a group deletes all items in that group.

```
delete(grp)
```

Find all remaining valid objects.

```
allOPC = opcfind
```

```
allOPC =
    [1x1 opcda]
```

Using the `delete` function with the object will remove the object from the OPC engine but not from the MATLAB® workspace. To remove an object from the MATLAB workspace use the `clear` function.

Display the current workspace.

```
whos
```

```
  Name            Size              Bytes  Class      Attributes

  allDA           1x1                 690  cell
  allOPC          1x1                 690  cell
  ans             0x0                   0  double
  da              1x1                 630  opcda
  foundGrp        1x1                 630  dagroup
  grp             1x1                 630  dagroup
  itm1            1x1                 630  daitem
  itm2            1x1                 630  daitem
```

Since an object was deleted, it is no longer valid.

```
grp
```

```
grp =
Invalid dagroup object.
This object should be removed from your workspace using CLEAR.
```

The items contained by that group are also invalid.

```
itm1
```

```
itm1 =
Invalid daitem object.
This object should be removed from your workspace using CLEAR.
```

Clear the associated variables.

```
clear grp itm1 itm2
```

Display the current workspace.

```
whos
```

```
  Name            Size             Bytes  Class      Attributes

  allDA           1x1                690  cell                 
  allOPC          1x1                690  cell                 
  ans             0x0                  0  double               
  da              1x1                630  opcda                
  foundGrp        1x1                630  dagroup              
```

To remove all OPC objects from the engine and to reset the toolbox to its initial state, use the `opcreset` function.

Note: Using the `opcreset` function will only delete objects from memory, not clear them from the MATLAB workspace.

```
opcreset
```

Verify that no objects remain.

```
allOPC = opcfind
```

```
allOPC =
     []
```

Variables associated with deleted objects still remain.

```
whos
```

```
  Name            Size             Bytes  Class      Attributes

  allDA           1x1                690  cell                 
  allOPC          0x0                  0  double               
  ans             0x0                  0  double               
  da              1x1                630  opcda                
  foundGrp        1x1                630  dagroup              
```

You can remove those variables using the `clear` function.

Read and Write Data to an OPC Data Access Server

This example shows you how to use synchronous read and write operations to exchange data with an OPC server.

PREREQUISITES:

- "Install a Simulation Server for OPC Examples" on page 21-2

Connect to Server and Create Objects

Create an `opcda` client and connect that client to the OPC server.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
```

Add a group to the client, and an item to the group.

```
grp = addgroup(da);
itm1 = additem(grp,'Random.Real8');
```

Perform Synchronous Read Operations

The default read operation gets values from the server cache.

```
r = read(itm1)
```

```
r = 

  struct with fields:

       ItemID: 'Random.Real8'
        Value: 0
      Quality: 'Bad: Out of Service'
    TimeStamp: [2016 8 30 11 55 24.4130]
        Error: ''
```

To force the server to read a value from the device, specify that option. This process can take a while if the OPC server is on the network or the device takes some time to produce a value.

```
r = read(itm1,'device')
```

```
r = 

  struct with fields:

       ItemID: 'Random.Real8'
        Value: 20.8848
      Quality: 'Good: Non-specific'
    TimeStamp: [2016 8 30 11 55 24.7220]
        Error: ''
```

Perform Synchronous Write Operations

Add a writable item to the group.

```
itm2 = additem(grp,'Bucket Brigade.Real8')
```

```
itm2 =

Summary of OPC Data Access Item Object: Bucket Brigade.Real8

   Object Parameters
      Parent        : Group0
      Access Rights : read/write

   Object Status
      Active        : on

   Data Parameters
      Data Type     : double
      Value         :
      Quality       : Bad: Out of Service
      Timestamp     :
```

Write the value 10 to the item.

```
write(itm2,10)
```

Read the value back into MATLAB.

```
r = read(itm2,'device')
```

```
r =

  struct with fields:

       ItemID: 'Bucket Brigade.Real8'
        Value: 10
      Quality: 'Good: Non-specific'
    TimeStamp: [2016 8 30 11 55 24.8520]
        Error: ''
```

Read From Multiple Items

You can read data from multiple items using the group object.

```
r = read(grp)
```

```
r =

  2×1 struct array with fields:

    ItemID
    Value
    Quality
    TimeStamp
    Error
```

Display individual item information by indexing.

```
r(1)
```

```
ans = 

  struct with fields:

       ItemID: 'Random.Real8'
        Value: 20.8848
      Quality: 'Good: Non-specific'
    TimeStamp: [2016 8 30 11 55 24.7220]
        Error: ''
```

Extract multiple values from item.

```
itmIDs = {r.ItemID}
vals = [r.Value]
```

```
itmIDs =

  1×2 cell array

    'Random.Real8'    'Bucket Brigade.Real8'


vals =

   20.8848   10.0000
```

Write to Multiple Items

Write to multiple items, passing the values for the items in the group as a cell array.

```
write(grp,{1.234,5.432})
```

```
Warning: One or more items could not be written.
    Random.Real8 returned 'The item's access rights do not allow the operation.'
```

The previous command returns a warning, because the first item does not allow you to write data to it. However, the second has the value 5.432 written. You can verify that be reading it.

```
r = read(itm2,'device')
```

```
r = 

  struct with fields:

       ItemID: 'Bucket Brigade.Real8'
        Value: 5.4320
```

```
      Quality: 'Good: Non-specific'
    TimeStamp: [2016 8 30 11 55 24.9070]
        Error: ''
```

Clean Up

Disconnect from the server and delete the client object.

```
disconnect(da)
delete(da)
```

Deleting the client object automatically deletes the group and item objects.

Log Data from an OPC Data Access Server

This example shows you how to configure and execute a logging session, and retrieve data from that logging session.

PREREQUISITES:

- “Install a Simulation Server for OPC Examples” on page 21-2

Create the OPC Object Hierarchy

Create a hierarchy of objects.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da,'CallbackTest');
additem(grp,'Random.Real8');
additem(grp,'Random.UInt2');
additem(grp,'Random.Real4');
```

Configure the Logging Duration

Set the group's `UpdateRate` value to `0.2` seconds, and the `RecordsToAcquire` property to `40`.

```
grp.UpdateRate = 0.2;
grp.RecordsToAcquire = 40;
```

Configure the Logging Destination

Configure the group to log data to disk and memory. Use a file in a temporary folder.

```
logFileName = fullfile(tempdir,'LoggingExample.olf');
grp.LoggingMode = 'disk&memory';
grp.LogFileName = logFileName;
grp.LogToDiskMode = 'overwrite';
```

The disk file name is `LoggingExample.olf`. If the file name exists, the toolbox engine overwrites the file.

Start the Logging Task

Start the logging task on the group object. Wait two seconds and show the last acquired value.

```
start(grp)
pause(2)
sPeek = peekdata(grp,1)
```

```
sPeek = 

    LocalEventTime: [2016 4 12 13 49 24.9080]
             Items: [3×1 struct]
```

Display the item ID and values

```
disp({sPeek.Items.ItemID;sPeek.Items.Value});
```

```
    'Random.Real8'    'Random.UInt2'    'Random.Real4'
    [  8.5714e+03]    [        9961]    [  1.9025e+04]
```

Wait for the object to complete logging before continuing with the example.

```
wait(grp)
```

Retrieve the Data

Retrieve the first 20 acquired records into a structure.

```
sFirst = getdata(grp,20);
```

The `getdata` function removes the records from the toolbox engine. Examine the available records using the `RecordsAvailable` property of the group.

```
recAvail = grp.RecordsAvailable
```

```
recAvail =

    20
```

Retrieve the balance of the records into separate arrays, converting all values to double-precision floating point numbers.

```
[exItmId,exVal,exQual,exTStamp,exEvtTime] = getdata(grp, ...
    recAvail,'double');
```

Examine the contents of the workspace.

```
whos ex*
```

```
  Name                Size              Bytes  Class     Attributes

  exEvtTime          20x1                 160  double              
  exItmId             1x3                 408  cell                
  exQual             20x3                8880  cell                
  exTStamp           20x3                 480  double              
  exVal              20x3                 480  double              
```

Retrieve data from disk for a specific item, using the 'itemids' filter.

```
sReal8Disk = opcread(logFileName,'itemids','Random.Real8')
```

```
sReal8Disk = 

40×1 struct array with fields:

    LocalEventTime
    Items
```

Examine the second record.

```
sReal8Disk(2).Items
```

```
ans = 

       ItemID: 'Random.Real8'
        Value: 1.4955e+04
      Quality: 'Good: Non-specific'
    TimeStamp: [2016 4 12 13 49 24.3890]
```

Clean Up

Disconnect and delete OPC objects from the toolbox engine.

```
disconnect(da)
delete(da)
delete(logFileName)
```

Deleting the client object also deletes the group and item objects.

View the OPC Event Log

This example shows you how to examine the OPC event log after a logging task.

PREREQUISITES:

- "Install a Simulation Server for OPC Examples" on page 21-2

Step 1: Configure OPC Objects

Create the client, connect, and create associated objects for a logging task.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da,'CallbackTest');
additem(grp,'Triangle Waves.Real8');
```

Step 2: Configure and Execute a Logging Task

Configure the group to log only 10 records, then start the task and wait for it to complete.

```
grp.RecordsToAcquire = 10;
start(grp)
wait(grp)
```

Step 3: View the Event Log

Access the `EventLog` property of the client object.

```
events = da.EventLog
```

```
events = 

  1×2 struct array with fields:

    Type
    Data
```

The execution of the group logging task generated two events: `start` and `stop`. The value of the `EventLog` property is a 1-by-2 array of event structures.

List the events that are recorded in the `EventLog` property, by examining the contents of the `Type` field.

```
{events.Type}
```

```
ans =

  1×2 cell array

    {'Start'}    {'Stop'}
```

Access the `Data` field to get information about the `stop` event.

```
stopdata = events(2).Data
```

```
stopdata = 

  struct with fields:

     LocalEventTime: [2020 10 19 11 38 3.8710]
          GroupName: 'CallbackTest'
    RecordsAcquired: 10
```

Calculate the time between the `stop` event and the `start` event.

```
waitDuration = datetime(events(2).Data.LocalEventTime)...
               - datetime(events(1).Data.LocalEventTime);
waitSeconds = seconds(waitDuration)
```

```
waitSeconds = 

    5.3740
```

Note: `waitSeconds` is not necessarily the time between the first and last sample in the logged data set. The `LocalEventTime` property is the time that MATLAB® processed the event received from the server; there can be some delay between the server sending the notification and MATLAB processing it. You should consult the `TimeStamp` property of the logged data for accurate time information related to the data.

Step 4: Clean Up

Disconnect the client from the server and remove OPC objects from memory when you no longer need them. Deleting the client object also deletes the group and item objects.

```
disconnect(da)
delete(da)
```

Monitor Logging Progress with Callbacks

This example shows you how to use callbacks to monitor an OPC Data Access logging task.

Use callbacks to log or report events in a logging task, to update graphical user interfaces to show status of logging, or to graphically display logged data during the logging task.

PREREQUISITES:

- "Install a Simulation Server for OPC Examples" on page 21-2

Step 1: Configure OPC Objects

Create the client, connect, and create associated objects for a logging task.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da,'CallbackTest');
additem(grp,{'Random.Real8','Saw-toothed Waves.UInt2'});
```

Step 2: Configure the Logging Task Properties

Set the group to acquire 20 records at 0.5 second intervals.

```
grp.RecordsToAcquire = 20;
grp.UpdateRate = 0.5;
disp(grp)
```

```
Summary of OPC Data Access Group Object: CallbackTest

   Object Parameters
      Group Type   : private
      Item         : 2-by-1 daitem object
      Parent       : localhost/Matrikon.OPC.Simulation.1
      Update Rate  : 0.5
      Deadband     : 0%

   Object Status
      Active       : on
      Subscription : on
      Logging      : off

   Logging Parameters
      Records      : 20
      Duration     : at least 10 seconds
      Logging to   : memory
      Status       : Waiting for START.
                     0 records available for GETDATA/PEEKDATA
```

Step 3: Configure the Callbacks

Use the default callback, opccallback, to display the start event (`StartFcn` property), the stop event (`StopFcn` property), and when each consecutive 5 records have been acquired (`RecordsAcquiredFcn` and `RecordsAcquiredFcnCount` properties).

```
grp.StartFcn = @opccallback;
grp.StopFcn = @opccallback;
grp.RecordsAcquiredFcn = @opccallback;
grp.RecordsAcquiredFcnCount = 5;
```

Step 4: Start the Logging Task

Start the logging task, and wait for it to complete.

```
start(grp)
wait(grp)
```

```
OPC Start event occurred at local time 14:22:38
    Group 'CallbackTest': 0 records acquired.


OPC RecordsAcquired event occurred at local time 14:22:41
    Group 'CallbackTest': 5 records acquired.


OPC RecordsAcquired event occurred at local time 14:22:44
    Group 'CallbackTest': 10 records acquired.


OPC RecordsAcquired event occurred at local time 14:22:47
    Group 'CallbackTest': 15 records acquired.
```

Step 5: Clean Up OPC Objects

Disconnect the client from the server and remove OPC objects from memory when you no longer need them. Deleting the client object also deletes the group and item objects.

```
disconnect(da)
delete(da)
```

Update MATLAB Plots While Logging OPC Data

This example shows you how to use a custom callback to plot data acquired during an OPC logging task.

The example makes use of the display_opcdata function, which plots recently acquired data in a figure window.

PREREQUISITES:

- "Install a Simulation Server for OPC Examples" on page 21-2

Step 1: Create the OPC Object Hierarchy

Create a hierarchy of OPC objects.

```
da = opcda('localhost','Matrikon.OPC.Simulation.1');
connect(da);
grp = addgroup(da,'CallbackTest');
additem(grp,'Triangle Waves.Real8');
additem(grp,'Saw-toothed Waves.UInt2');
```

Step 2: Configure Property Values

Configure the logging task to acquire 200 records at 0.1 second intervals.

```
grp.RecordsToAcquire = 200;
grp.UpdateRate = 0.1;
```

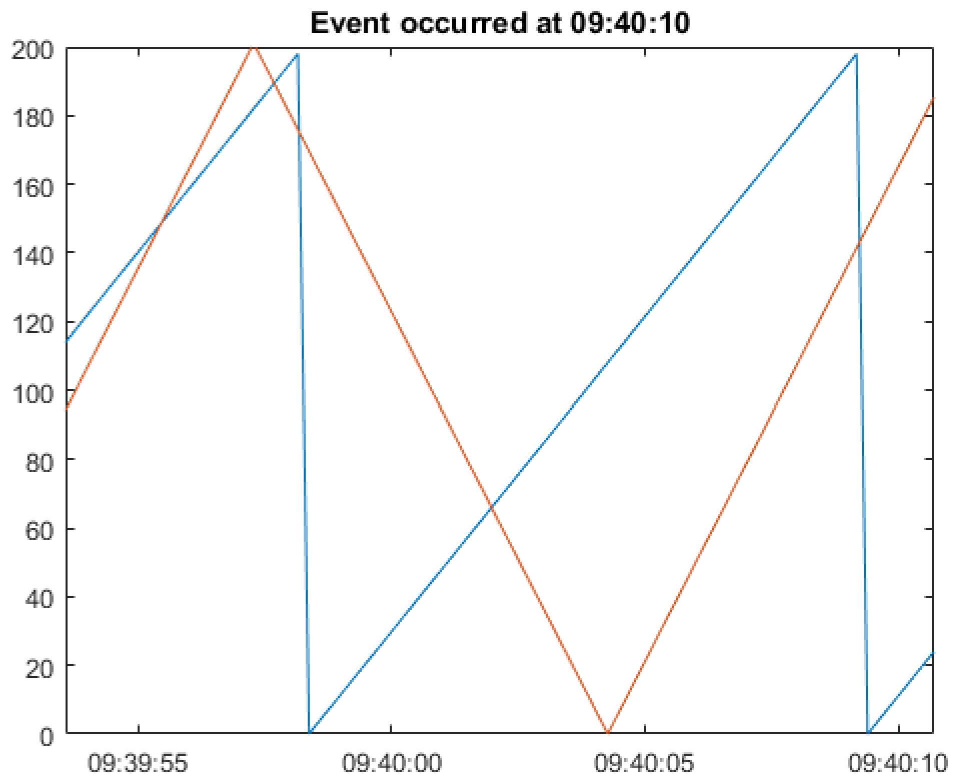
Specify the display_opcdata function as the `RecordsAcquiredFcn` callback, which must be called after each 10 records are acquired.

```
grp.RecordsAcquiredFcnCount = 10;
grp.RecordsAcquiredFcn = @display_opcdata;
```

Step 3: Acquire Data

Start the group object. After every 10 records are acquired, the object executes the `display_opcdata` callback function. This callback function plots the most recently acquired records logged to the memory buffer.

```
start(grp)
wait(grp)
```



Step 4: Clean Up

Always remove OPC objects from memory when you no longer need them.

```
delete(da)
```

Deleting the client object disconnects the client from the server, and deletes the group and item objects.

Locate and Browse OPC Historical Data Access Servers

This example shows how to browse the network for OPC Historical Data Access servers, and query the server name space for server items and their properties.

PREREQUISITES:

- “Install a Simulation Server for OPC Examples” on page 21-2

Step 1: Browse the Network for OPC HDA Servers

You use the `opchdaserverinfo` function to query a host on the network for available OPC Historical Data Access servers. This example uses the local host.

```
hostInfo = opchdaserverinfo('localhost')
```

```
hostInfo =

OPC HDA Server Information object:
                 Host: localhost
             ServerID: Matrikon.OPC.Simulation.1
          Description: MatrikonOPC Server for Simulation and Testing
     HDASpecification: HDA1
```

Find the server info entry with a description starting with Matrikon.

```
hIndex = findDescription(hostInfo,'Matrikon')
hostInfo(hIndex)
```

```
hIndex =

     1


ans =

OPC HDA Server Information object:
                 Host: localhost
             ServerID: Matrikon.OPC.Simulation.1
          Description: MatrikonOPC Server for Simulation and Testing
     HDASpecification: HDA1
```

Step 2: Construct a Client Object and Connect to the Server

Use the `ServerInfo` object returned in the previous step to construct a client object.

```
hdaObj = opchda(hostInfo(hIndex));
```

You can also specify the host name and server ID directly.

```
hdaObj = opchda('localhost','Matrikon.OPC.Simulation.1')
```

```
hdaObj =
OPC HDA Client localhost/Matrikon.OPC.Simulation.1:
               Host: localhost
           ServerID: Matrikon.OPC.Simulation.1
            Timeout: 10 seconds

             Status: disconnected

         Aggregates: -- (client is disconnected)
     ItemAttributes: -- (client is disconnected)
```

Connect the client to the server.

```
connect(hdaObj);
```

Step 3: Retrieve the Server Name Space

Retrieve the name space of the server.

```
ns = getNameSpace(hdaObj)
```

```
ns = 

4×1 struct array with fields:

    Name
    FullyQualifiedID
    NodeType
    Nodes
```

Each element of the structure is a node in the server name space.

```
ns(1)
```

```
ans = 

                Name: 'Simulation Items'
    FullyQualifiedID: 'Simulation Items¥'
            NodeType: 'branch'
               Nodes: [8×1 struct]
```

Step 4: Find Items in the Name Space

Use the `serveritems` function to find all items in the name space containing the string `Real`.

```
realItems = serveritems(ns,'*Real*')
```

```
realItems = 

    'Bucket Brigade.ArrayOfReal8'
    'Bucket Brigade.Real4'
    'Bucket Brigade.Real8'
    'Random.ArrayOfReal8'
    'Random.Real4'
    'Random.Real8'
    'Read Error.ArrayOfReal8'
    'Read Error.Real4'
    'Read Error.Real8'
    'Saw-toothed Waves.Real4'
    'Saw-toothed Waves.Real8'
    'Square Waves.Real4'
    'Square Waves.Real8'
    'Triangle Waves.Real4'
    'Triangle Waves.Real8'
    'Write Error.ArrayOfReal8'
    'Write Error.Real4'
    'Write Error.Real8'
    'Write Only.ArrayOfReal8'
    'Write Only.Real4'
    'Write Only.Real8'
```

Step 5: Query Server Item Attributes

Examine the current normal maximum value of the tenth item found.

```
maxVal = readItemAttributes(hdaObj,realItems{10},hdaObj.ItemAttributes.NORMAL_MAXIMUM,now,now)
```

```
Warning: Saw-toothed Waves.Real4: No history available for attribute.

maxVal = 

         ItemID: 'Saw-toothed Waves.Real4'
    AttributeID: 11
      Timestamp: 7.3643e+05
          Value: 100
```

The warning indicates that the item has not yet been stored in the historian database, but the preconfigured item attributes are being returned.

Step 6: Clean Up OPC Objects

Disconnect the client from the server and remove OPC objects from memory when you no longer need them. Deleting the client object also deletes the group and item objects.

```
disconnect(hdaObj)
delete(hdaObj)
```

Acquire Data from an OPC Historical Data Access Server

This example shows you how to acquire data from an OPC Historical Data Access (HDA) server.

PREREQUISITES:

- “Install a Simulation Server for OPC Examples” on page 21-2

Start Historical Data Logging on the Server

NOTE: *You do not normally need to execute this step on a production server.*

This example uses a simulation server that only logs historical data for items that are subscribed using an OPC Data Access client. Load the client object from a MAT file and reconnect the client.

```
daObjs = load('opcdemoHDAConfigure.mat');
connect(daObjs.opcdemoHDAConfigure);
```

Wait a while for the server to log some data.

```
pause(10);
```

Create an OPC HDA Client Object

Create an OPC HDA Client associated with the OPC HDA server.

```
hdaObj = opchda('localhost','matrikon.OPC.Simulation')
```

```
hdaObj =

OPC HDA Client localhost/matrikon.OPC.Simulation:
               Host: localhost
           ServerID: matrikon.OPC.Simulation
            Timeout: 10 seconds

             Status: disconnected

         Aggregates: -- (client is disconnected)
     ItemAttributes: -- (client is disconnected)
```

The client object manages the connection with the server, allows you to retrieve information about the server, browse the server name space, and to read data stored on the server.

At this point, the client is not yet connected to the server. Connect the client to the server.

```
connect(hdaObj);
```

To confirm that the client is connected, display the client `Status` property.

```
hdaObj.Status
```

```
ans =
connected
```

Define Items of Interest

This example uses the `Real8` items from `Saw-toothed Waves` and the `Real8` and `UInt2` items from `Random`. Make a cell array of item names for ease-of-use.

```
itmIDs = {'Saw-toothed Waves.Real8', ...
    'Random.Real8', ...
    'Random.UInt2'};
```

Read Raw Data from the Server

Read the raw data values from the historical server over the past day.

```
data = readRaw(hdaObj,itmIDs,now-1,now)
```

```
data = 

1-by-3 OPC HDA Data object:

             ItemID                Value           Start TimeStamp           End TimeStamp
    -----------------------  ---------------  -----------------------  -----------------------
    Saw-toothed Waves.Real8  9 double values  2016-04-12 16:30:17.662  2016-04-12 16:30:25.776
    Random.Real8             9 double values  2016-04-12 16:30:17.662  2016-04-12 16:30:25.776
    Random.UInt2             9 uint16 values  2016-04-12 16:30:17.662  2016-04-12 16:30:25.776

Use the showValues method to display all values.
```

Note: The Matrikon server retains only the last 200 simulated values for each item.

Display the values of the first data element.

```
showValues(data(1))
```

```
OPC HDA Data object for item Saw-toothed Waves.Real8:

           TIMESTAMP               VALUE       QUALITY
    =======================  =============  ==========
    2016-04-12 16:30:17.662       3.141593  Raw (Good)
    2016-04-12 16:30:18.677       6.283185  Raw (Good)
    2016-04-12 16:30:19.692       9.424778  Raw (Good)
    2016-04-12 16:30:20.707      12.566371  Raw (Good)
    2016-04-12 16:30:21.717      15.707963  Raw (Good)
    2016-04-12 16:30:22.732      18.849556  Raw (Good)
    2016-04-12 16:30:23.747      21.991149  Raw (Good)
    2016-04-12 16:30:24.761      25.132741  Raw (Good)
    2016-04-12 16:30:25.776      28.274334  Raw (Good)
```

Read Processed Data from the Server

Query the `Aggregates` property of the HDA Client object to find out what aggregate types the server supports.

```
hdaObj.Aggregates
```

```
ans =

OPC HDA Aggregate Types:
          Name         ID                                          Description
    -----------------  --  ---------------------------------------------------------------------
    INTERPOLATIVE      1   Retrieve interpolated values.
    TIMEAVERAGE        4   Retrieve the time weighted average data over the resample interval.
    MINIMUMACTUALTIME  7   Retrieve the minimum value in the resample interval and the timestamp
    MINIMUM            8   Retrieve the minimum value in the resample interval.
    MAXIMUMACTUALTIME  9   Retrieve the maximum value in the resample interval and the timestamp
    MAXIMUM            10  Retrieve the maximum value in the resample interval.
```

The Matrikon server supports the time weighted average value, so we will use that aggregate type on 10 seconds of data for the last 1 minute. Note below how the `Aggregates` property can be used to specify the aggregate type.

```
pData = readProcessed(hdaObj,itmIDs,hdaObj.Aggregates.TIMEAVERAGE,10,now-1/24/60,now)'
```

```
pData =

1-by-3 OPC HDA Data object:

           ItemID                Value           Start TimeStamp           End TimeStamp
    -----------------------  ---------------  -----------------------  -----------------------
    Saw-toothed Waves.Real8  6 double values  2016-04-12 16:29:26.840  2016-04-12 16:30:16.840
    Random.Real8             6 double values  2016-04-12 16:29:26.840  2016-04-12 16:30:16.840
    Random.UInt2             6 uint16 values  2016-04-12 16:29:26.840  2016-04-12 16:30:16.840

Use the showValues method to display all values.
```

Display the values for the `Random.Real8` item.

```
itmInd = getIndexFromID(pData,'Random.Real8');
showValues(pData(itmInd))
```

```
OPC HDA Data object for item Random.Real8:

           TIMESTAMP               VALUE              QUALITY
    =======================  =============  ======================
    2016-04-12 16:29:26.840    5073.986117  Calculated (Uncertain)
    2016-04-12 16:29:36.840    5073.986074  Calculated (Uncertain)
    2016-04-12 16:29:46.840    5073.986105  Calculated (Uncertain)
    2016-04-12 16:29:56.840    5073.986137  Calculated (Uncertain)
    2016-04-12 16:30:06.840    5073.986227  Calculated (Uncertain)
    2016-04-12 16:30:16.840    7322.794889  Calculated (Uncertain)
```

The last value has a quality of `'Uncertain'` because the time interval is not a complete 10 seconds.

Clean Up

When you have finished with the OPC objects, delete them from the OPC engine. Although deleting an HDA Client object automatically disconnects the object from the server, this example explicitly shows it.

```
disconnect(hdaObj)
delete(hdaObj)
disconnect(daObjs.opcdemoHDAConfigure);
delete(daObjs.opcdemoHDAConfigure);
```

The client object is now invalid.

```
isvalid(hdaObj)
```

```
ans =

   0
```

Visualize and Preprocess OPC HDA Data

This example shows you how to work with OPC HDA Data objects.

You create OPC HDA Data objects when you read data from an OPC Historical Data Access (HDA) server. OPC HDA Data objects allow you to store, visualize and manipulate historical data before converting that data to built-in data types for further processing in MATLAB.

For more information on generating OPC HDA Data objects, see the example “Acquire Data from an OPC Historical Data Access Server” on page 21-33.

Load Sample OPC HDA Data

Load the sample data into the workspace.

```
load opcdemoHDAData
```

Display OPC HDA Data Objects

Examine the workspace to see the loaded variables.

```
whos
```

```
  Name                Size            Bytes  Class           Attributes

  hdaDataSmall        1x2               356  opc.hda.Data
  hdaDataVis          1x2              9088  opc.hda.Data
```

Display a summary of the data contained in `hdaDataVis`.

```
hdaDataVis
```

```
hdaDataVis =

1-by-2 OPC HDA Data object:

        ItemID            Value            Start TimeStamp           End TimeStamp           
    --------------  -----------------  -----------------------  -----------------------  --------
    Example.Item.1  361 double values  2010-05-12 08:15:00.000  2010-05-12 09:15:00.000  1 unique
    Example.Item.2  11 double values   2010-05-12 08:30:00.000  2010-05-12 09:30:00.000  2 unique
```

The data object contains two items. The first element `Example.Item.1` contains 361 values and one unique quality, while the second has 11 values and two unique qualities.

Examine the second element in more detail using the `showValues` function.

```
showValues(hdaDataVis(2))
```

```
OPC HDA Data object for item Example.Item.2:

           TIMESTAMP                VALUE          QUALITY
    =======================  =============  =================
    2010-05-12 08:30:00.000      -0.500000  Raw (Good)
    2010-05-12 08:36:00.000      -0.250000  Raw (Good)
    2010-05-12 08:42:00.000       0.000000  Raw (Good)
    2010-05-12 08:48:00.000       0.250000  Raw (Good)
    2010-05-12 08:54:00.000       0.500000  Calculated (Good)
    2010-05-12 09:00:00.000       0.500000  Calculated (Good)
    2010-05-12 09:06:00.000       0.400000  Calculated (Good)
    2010-05-12 09:12:00.000       0.300000  Raw (Good)
    2010-05-12 09:18:00.000       0.200000  Raw (Good)
    2010-05-12 09:24:00.000       0.100000  Raw (Good)
    2010-05-12 09:30:00.000       0.000000  Raw (Good)
```

Change the Date Display Format

Get the current date display format using `opc.getDateDisplayFormat`.

```
origFormat = opc.getDateDisplayFormat;
```

Change the display format to standard US date format and display the value again.

```
opc.setDateDisplayFormat('mm/dd/yyyy HH:MM AM');
showValues(hdaDataVis(2))
```

```
OPC HDA Data object for item Example.Item.2:

         TIMESTAMP           VALUE          QUALITY
    ===================  =============  =================
    05/12/2010  8:30 AM      -0.500000  Raw (Good)
    05/12/2010  8:36 AM      -0.250000  Raw (Good)
    05/12/2010  8:42 AM       0.000000  Raw (Good)
    05/12/2010  8:48 AM       0.250000  Raw (Good)
    05/12/2010  8:54 AM       0.500000  Calculated (Good)
    05/12/2010  9:00 AM       0.500000  Calculated (Good)
    05/12/2010  9:06 AM       0.400000  Calculated (Good)
    05/12/2010  9:12 AM       0.300000  Raw (Good)
    05/12/2010  9:18 AM       0.200000  Raw (Good)
    05/12/2010  9:24 AM       0.100000  Raw (Good)
    05/12/2010  9:30 AM       0.000000  Raw (Good)
```

Reset the display format to the default.

```
opc.setDateDisplayFormat('default')
```

```
ans =

yyyy-mm-dd HH:MM:SS.FFF
```

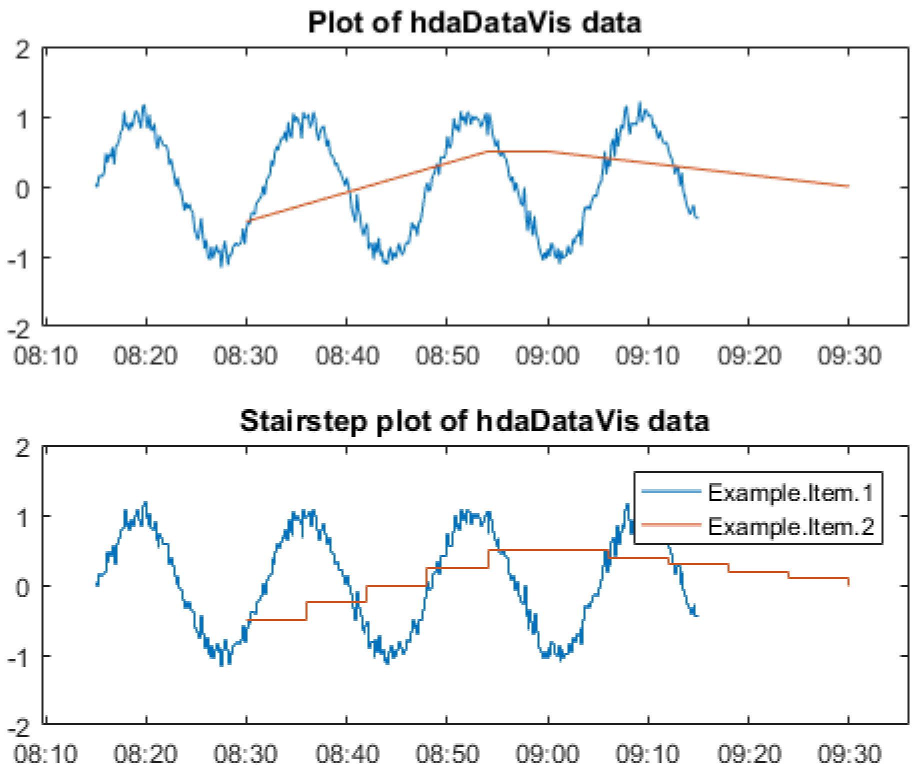
Reset the display format to the original value.

```
opc.setDateDisplayFormat(origFormat);
```

Visualize OPC HDA Data

Visualize OPC HDA Data using the `plot` and `stairs` functions on the data object.

```
axH1 = subplot(2,1,1);
plot(hdaDataVis);
title('Plot of hdaDataVis data');
axH2 = subplot(2,1,2);
stairs(hdaDataVis);
title('Stairstep plot of hdaDataVis data');
legend show
```



Resample OPC HDA Data

Examine a small data set. This data set is intentionally small to show the concept of resampling.

```
hdaDataSmall
```

```
hdaDataSmall =

1-by-2 OPC HDA Data object:

        ItemID            Value           Start TimeStamp           End TimeStamp            Qu
    ---------------  ---------------  -----------------------  -----------------------  ---------
    Example.ItemR.1  5 double values  2010-06-01 09:30:00.000  2010-06-01 09:31:00.000  1 unique
    Example.ItemR.2  3 double values  2010-06-01 09:30:00.000  2010-06-01 09:31:00.000  1 unique
```

Display the data from each item individually. You cannot display the items in one table because their time stamps are not the same.

```
showValues(hdaDataSmall(1))
showValues(hdaDataSmall(2))
```

```
OPC HDA Data object for item Example.ItemR.1:

         TIMESTAMP               VALUE       QUALITY
    =======================  =============  ==========
    2010-06-01 09:30:00.000       0.000000  Raw (Good)
    2010-06-01 09:30:15.000       1.000000  Raw (Good)
    2010-06-01 09:30:30.000       2.000000  Raw (Good)
    2010-06-01 09:30:45.000       1.000000  Raw (Good)
    2010-06-01 09:31:00.000       0.000000  Raw (Good)


OPC HDA Data object for item Example.ItemR.2:

         TIMESTAMP               VALUE       QUALITY
    =======================  =============  ==========
    2010-06-01 09:30:00.000       1.000000  Raw (Good)
    2010-06-01 09:30:30.000       2.000000  Raw (Good)
    2010-06-01 09:31:00.000       3.000000  Raw (Good)
```

Attempt to convert the data to a double array. The conversion will fail.

```
try
    vals = double(hdaDataSmall);
catch exc
    disp(exc.message)
end
```

```
Conversion to double failed. All elements of the OPC HDA Data object must have the same time star
Consider using 'TSUNION' or 'RESAMPLE' on the Data object.
```

The intersection of the items' time stamps results in a smaller, regularly sampled data set.

```
hdaDataIntersect = hdaDataSmall.tsintersect
```

```
hdaDataIntersect = 

1-by-2 OPC HDA Data object:

        ItemID           Value          Start TimeStamp           End TimeStamp           Qu
    ---------------  ---------------  -----------------------  -----------------------  ---------
    Example.ItemR.1  3 double values  2010-06-01 09:30:00.000  2010-06-01 09:31:00.000  1 unique
    Example.ItemR.2  3 double values  2010-06-01 09:30:00.000  2010-06-01 09:31:00.000  1 unique

Use the showValues function to display all values.
```

Show these values together. You can do this because the time stamps are now regularly sampled.

```
showValues(hdaDataIntersect)
```

```
OPC HDA Data object array:

                  TIMESTAMP   Example.ItemR.1  Example.ItemR.2
```

```
    =======================  ===============  ===============
    2010-06-01 09:30:00.000         0.000000         1.000000
    2010-06-01 09:30:30.000         2.000000         2.000000
    2010-06-01 09:31:00.000         0.000000         3.000000
```

Convert the data object into a double array.

```
vals = double(hdaDataIntersect)
```

```
vals =

     0     1
     2     2
     0     3
```

Use `tsunion` to return the union of time series in a Data object. New values are interpolated using the method supplied (or linear interpolation if no method is supplied).

```
hdaDataUnion = hdaDataSmall.tsunion
showValues(hdaDataUnion)
```

```
hdaDataUnion =

1-by-2 OPC HDA Data object:

        ItemID           Value          Start TimeStamp          End TimeStamp            Qu
    ---------------  ---------------  -----------------------  -----------------------  ---------
    Example.ItemR.1  5 double values  2010-06-01 09:30:00.000  2010-06-01 09:31:00.000  1 unique
    Example.ItemR.2  5 double values  2010-06-01 09:30:00.000  2010-06-01 09:31:00.000  2 unique

Use the showValues function to display all values.

OPC HDA Data object array:

                  TIMESTAMP  Example.ItemR.1  Example.ItemR.2
    =======================  ===============  ===============
    2010-06-01 09:30:00.000         0.000000         1.000000
    2010-06-01 09:30:15.000         1.000000         1.500000
    2010-06-01 09:30:30.000         2.000000         2.000000
    2010-06-01 09:30:45.000         1.000000         2.500000
    2010-06-01 09:31:00.000         0.000000         3.000000
```

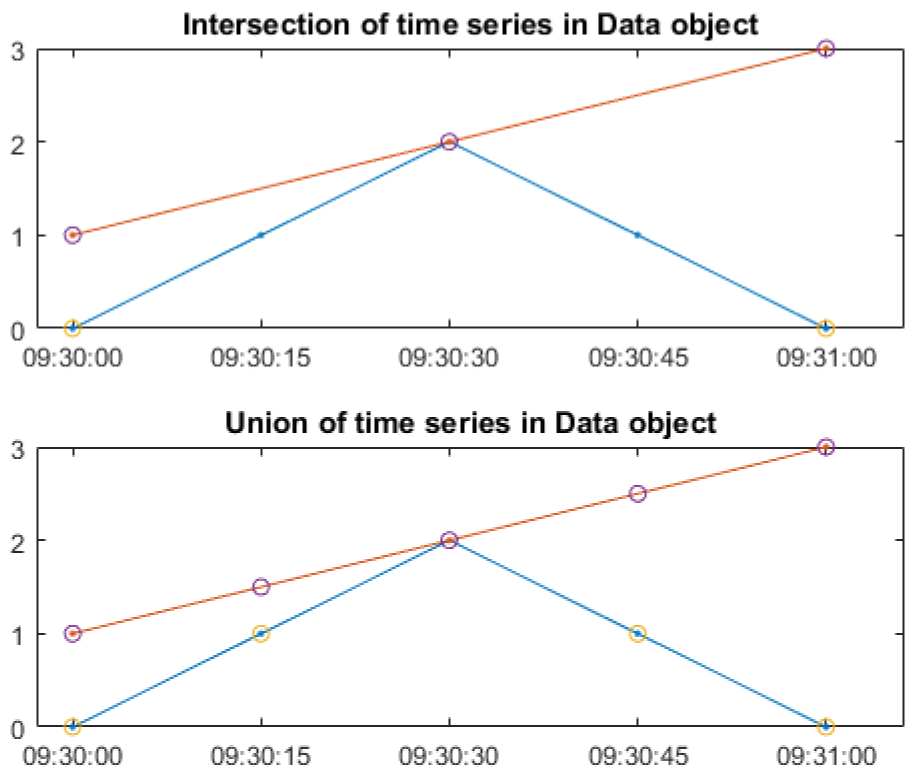
Note how the quality is set to "Interpolated" for those new values in `Example.ItemR.2`.

```
showValues(hdaDataUnion(2))
```

```
OPC HDA Data object for item Example.ItemR.2:

           TIMESTAMP            VALUE            QUALITY
    =======================  =============  ===================
    2010-06-01 09:30:00.000       1.000000  Raw (Good)
    2010-06-01 09:30:15.000       1.500000  Interpolated (Good)
    2010-06-01 09:30:30.000       2.000000  Raw (Good)
    2010-06-01 09:30:45.000       2.500000  Interpolated (Good)
    2010-06-01 09:31:00.000       3.000000  Raw (Good)
```

Plot the data with markers to show how the methods work.

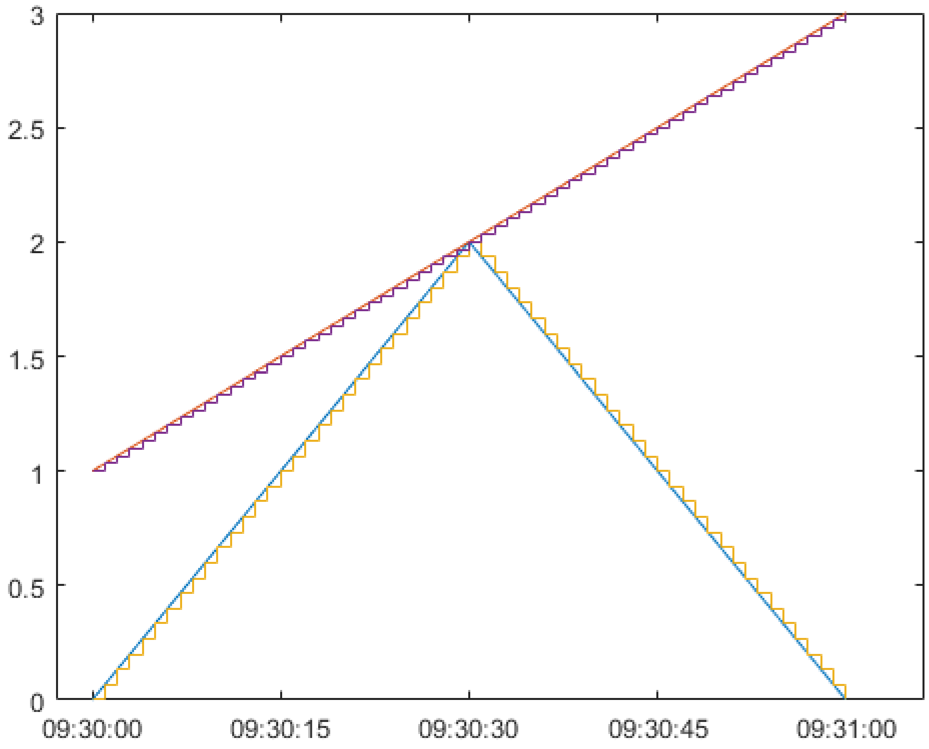
```
subplot(2,1,1);
plot(hdaDataSmall,'Marker','.');
hold all
plot(hdaDataIntersect,'Marker','o','LineStyle','none');
title('Intersection of time series in Data object');
subplot(2,1,2);
plot(hdaDataSmall,'Marker','.');
hold all
plot(hdaDataUnion,'Marker','o','LineStyle','none');
title('Union of time series in Data object');
```



Resample the small data set at specified time steps.

```
newTS = datenum(2010,6,1,9,30,[0:60]);
hdaDataResampled = resample(hdaDataSmall,newTS)
figure;
plot(hdaDataSmall);
hold all
stairs(hdaDataResampled);
```

```
hdaDataResampled =
1-by-2 OPC HDA Data object:

        ItemID            Value           Start TimeStamp           End TimeStamp           Qua
    ---------------  ----------------  -----------------------  -----------------------  --------
    Example.ItemR.1  61 double values  2010-06-01 09:30:00.000  2010-06-01 09:31:00.000  2 unique
    Example.ItemR.2  61 double values  2010-06-01 09:30:00.000  2010-06-01 09:31:00.000  2 unique

Use the showValues function to display all values.
```



Browse OPC UA Server Namespace

This example shows you how to find OPC Unified Architecture (UA) servers, connect to them, and browse their namespace to find nodes of interest.

To run this example in your MATLAB® session, you must install and start the Prosys OPC UA Simulation Server. For further information, see the Getting Started section of the Industrial Communication Toolbox™ documentation.

OPC UA servers structure available data through one or more namespaces, consisting of multiple connected Nodes. Each namespace has an Index uniquely identifying that namespace. The toolbox exposes two types of OPC UA Nodes: Object nodes, which help to organise data, and Variable nodes which store data in their Value property. Variables nodes may contain other Variable nodes as children.

All OPC UA servers must publish a Server node, containing information about the OPC UA server including capabilities of that server, available functionality of the server and other diagnostic information. The Server node must exist as namespace index 0, named 'Server'. This example will explore the ServerCapabilities node contained in the Server node of an example OPC UA server.

Explore Available OPC UA Servers on a Host

NOTE: This section of this example requires you to install the Local Discovery Service, and configure the Prosys OPC UA Simulation Server to register with the LDS. Instructions for how to do this are included in the Getting Started section of the Industrial Communication Toolbox documentation.

OPC UA servers may register with a Local Discovery Service on their host. The Local Discovery Service (LDS) publishes all available servers, as well as their unique "address" (or URL) for connecting to that server.

You can discover OPC UA servers available on a host using `opcuaserverinfo`. This example uses the local host.

```
serverList = opcuaserverinfo('localhost')
```

```
serverList = 

1×3 OPC UA ServerInfo array:
    index             Description                            Hostname                Port 
    -----  -----------------------------------  ---------------------------------  -----
      1    SimulationServer                     tmopti01win1064.dhcp.mathworks.com  53530
      2    UA Sample Server                     tmopti01win1064                     51210
      3    Quickstart Historical Access Server  tmopti01win1064                     62550
```

The list of servers shows the available OPC UA servers, and the hostname and port number on which you can connect to the server. You can find a specific server by searching the Description of the servers. Find the server containing the word "Simulation".

```
sampleServerInfo = findDescription(serverList, 'Simulation')
```

```
sampleServerInfo = 

OPC UA ServerInfo 'SimulationServer':
   Connection Information
    Hostname: 'tmopti01win1064.dhcp.mathworks.com'
        Port: 53530
```

Construct an OPC UA Client and Connect to the Server

In order to browse the server namespace, you need to construct an OPC UA Client and connect that client to the server. If you know the hostname and port of the OPC UA server, you could simply construct an OPC UA Client using the hostname and port arguments.

```
uaClient = opcua('localhost', 53530);
```

If you have previously discovered the server using the `opcuaserverinfo` command, you can construct the client directly from the `opcuaserverinfo` results.

```
uaClient = opcua(sampleServerInfo)
```

```
uaClient = 

OPC UA Client SimulationServer:
                     Hostname: tmopti01win1064.dhcp.mathworks.com
                         Port: 53530
                      Timeout: 10

                       Status: Disconnected
```

Initially the client is disconnected from the server, and shows a brief summary of the client properties. You know that the client is disconnected by querying the Status property, or calling the `isConnected` function.

```
status = uaClient.Status

isConnected(uaClient)
```

```
status = 

    'Disconnected'


ans =

  logical

   0
```

Once you connect the client to the server, additional properties from the server are displayed.

```
connect(uaClient)
uaClient
```

```
uaClient = 

OPC UA Client SimulationServer:
```

```
                   Hostname: tmopti01win1064.dhcp.mathworks.com
                       Port: 53530
                    Timeout: 10

                     Status: Connected

                ServerState: Running

              MinSampleRate: 0 sec
        MaxHistoryReadNodes: 0
    MaxHistoryValuesPerNode: 0
               MaxReadNodes: 0
              MaxWriteNodes: 0
```

The display shows that the client Status is now 'Connected', the server is in the 'Running' state, and the client stores information regarding server limits. In this case, all limits are set to zero, indicating that there is no server-wide limit for sample rates, maximum nodes or values for read operations on the Sample Server.

Browsing the Server Namespace

The server namespace is incrementally retrieved directly into the OPC UA Client variable in MATLAB. You access the top level of the server namespace using the `Namespace` property. This property stores OPC UA Nodes. Each node can contain one or more Children, which are themselves nodes.

```
topNodes = uaClient.Namespace
```

```
topNodes = 

1x6 OPC UA Node array:
    index           Name            NsInd        Identifier         NodeType  Children
    -----  -----------------------  -----  -----------------------  --------  --------
      1    Server                   0      2253                     Object    12
      2    MyObjects                2      MyObjectsFolder          Object    1
      3    StaticData               3      StaticData               Object    9
      4    NonUaNodeComplianceTest  4      NonUaNodeComplianceTest  Object    33
      5    Simulation               5      85/0:Simulation          Object    7
      6    MyBigNodeManager         6      MyBigNodeManager         Object    1000
```

The node named `'Server'` contains 12 children.

You can search the namespace using indexing into the Children property of available nodes. For example, to find the ServerCapabilities node, you can query the Children of the Server node.

```
serverChildren = topNodes(1).Children
```

```
serverChildren = 

1x12 OPC UA Node array:
    index         Name          NsInd  Identifier  NodeType  Children
    -----  ------------------  -----  ----------  --------  --------
      1    ServerStatus        0      2256        Variable  6
      2    ServerCapabilities  0      2268        Object    14
      3    ServerDiagnostics   0      2274        Object    4
      4    VendorServerInfo    0      2295        Object    0
```

```
     5    ServerRedundancy     0      2296        Object    5
     6    Namespaces           0      11715       Object    1
     7    ServerConfiguration  0      12637       Object    5
     8    NamespaceArray       0      2255        Variable  0
     9    Auditing             0      2994        Variable  0
    10    ServerArray          0      2254        Variable  0
    11    EstimatedReturnTime  0      12885       Variable  0
    12    ServiceLevel         0      2267        Variable  0
```

The ServerCapabilities node is the second node in the list.

```
serverCapabilities = serverChildren(2)
```

```
serverCapabilities = 

OPC UA Node object:
                      Name: ServerCapabilities
               Description: Describes capabilities supported by the server.
            NamespaceIndex: 0
                Identifier: 2268
                  NodeType: Object

                    Parent: Server
                  Children: 14 nodes.
```

Searching for Nodes in the Namespace

You can search for nodes from a Node variable, or from the Namespace property directly. To find the 'ServerCapabilities' node without indexing into the `Namespace` property, use `findNodeByName`. To avoid the search finding all instances of nodes containing the word 'ServerCapabilities' you use the `'-once'` parameter.

```
serverCapabilities = findNodeByName(topNodes, 'ServerCapabilities', '-once')
```

```
serverCapabilities = 

OPC UA Node object:
                      Name: ServerCapabilities
               Description: Describes capabilities supported by the server.
            NamespaceIndex: 0
                Identifier: 2268
                  NodeType: Object

                    Parent: Server
                  Children: 14 nodes.
```

To find all nodes containing the word 'Double' in the Name, query all topNodes using the `'-partial'` parameter. Note that this search will load the entire namespace into MATLAB, so use this search method with caution.

```
doubleNodes = findNodeByName(topNodes, 'Double', '-partial')
```

```
doubleNodes = 
1x6 OPC UA Node array:
    index          Name           NsInd       Identifier        NodeType  Children
    -----  ---------------------  -----  ---------------------  --------  --------
      1    Double                 4      Double                 Variable  0
      2    DoubleAnalogItemArray  3      DoubleAnalogItemArray  Variable  3
      3    DoubleAnalogItem       3      DoubleAnalogItem       Variable  3
      4    DoubleDataItem         3      DoubleDataItem         Variable  1
      5    DoubleArray            3      DoubleArray            Variable  0
      6    Double                 3      Double                 Variable  0
```

Understanding the NodeType

Nodes have a NodeType which describes whether that node is simply an organisational unit (an Object NodeType) or contains data that can be read or written (a Variable NodeType). An example of an Object node is tha ServerCapabilities node shown above. You cannot read data from an Object node. In this example, `doubleNodes` contains no Object nodes, and 6 Variable nodes.

```
allNodeTypes = {doubleNodes.NodeType}
```

```
allNodeTypes =

  1×6 cell array

  Columns 1 through 4

    {'Variable'}    {'Variable'}    {'Variable'}    {'Variable'}

  Columns 5 through 6

    {'Variable'}    {'Variable'}
```

Variable NodeTypes may contain Children - A NodeType of Variable does not guarantee that the node contains no Children. The second node listed is a variable node (and so its Value can can be read) but also has children (which can be read individually). For information on reading values from a node, see `readValue`.

Understanding Variable NodeType Properties

A Variable node has additional properties describing the data stored in the Variable node, including the server data type and access permissions for that node. To view these properties, display a Variable node.

```
doubleNodes(2)
```

```
ans =

OPC UA Node object:
                      Name: DoubleAnalogItemArray
               Description:
            NamespaceIndex: 3
                Identifier: DoubleAnalogItemArray
                  NodeType: Variable
                Parent: AnalogItemArrays
              Children: 3 nodes.

        ServerDataType: Double
    AccessLevelCurrent: read/write
    AccessLevelHistory: none
           Historizing: 0
```

This node has a `ServerDataType` of 'Double', and allows reading and writing of the Current value (`AccessLevelCurrent` property) but supports no historical data reading (`AccessLevelHistory`). The server is not Historizing this node, as evidenced by the `Historizing` property.

Some properties, such as `ServerValueRank`, and `ServerArrayDimensions` are not shown in the display of a node, but can be queried through the respective property. See help on these properties for further information.

```
doubleNodes(2).ServerArrayDimensions
```

```
ans =

  uint32

   0
```

Constructing Nodes Directly

Nodes are defined uniquely by their NamespaceIndex and their Identifier. You can construct a known node without browsing the `Namespace` property using the `opcuanode` function. For example, to construct the ServerCapabilities node directly you can use the NamespaceIndex 0 and Identifier 2268 (all OPC UA servers must publish a ServerCapabilities node with this NamespaceIndex and Identifier).

```
capabilitiesNode = opcuanode(0, 2268, uaClient)
```

```
capabilitiesNode = 

OPC UA Node object:
                  Name: ServerCapabilities
           Description: Describes capabilities supported by the server.
        NamespaceIndex: 0
            Identifier: 2268
              NodeType: Object

              Children: 14 nodes.
```

Note that nodes constructed using `opcuanode` have no Parent property.

```
capabilitiesNode.Parent
```

```
ans = 

Empty OPC UA Node object.
```

However their Children are automatically retrieved if the node is associated with a connected OPC UA Client.

```
capabilitiesNode.Children
```

```
ans = 

1x14 OPC UA Node array:
    index              Name              NsInd  Identifier  NodeType  Children
    -----  ----------------------------  -----  ----------  --------  --------
      1    ModellingRules                0      2996        Object    6
      2    AggregateFunctions            0      2997        Object    14
      3    HistoryServerCapabilities     0      11192       Object    15
      4    OperationLimits               0      11704       Object    12
      5    LocaleIdArray                 0      2271        Variable  0
      6    MinSupportedSampleRate        0      2272        Variable  0
      7    MaxQueryContinuationPoints    0      2736        Variable  0
      8    MaxByteStringLength           0      12911       Variable  0
      9    ServerProfileArray            0      2269        Variable  0
     10    MaxHistoryContinuationPoints  0      2737        Variable  0
     11    SoftwareCertificates          0      3704        Variable  0
     12    MaxStringLength               0      11703       Variable  0
     13    MaxBrowseContinuationPoints   0      2735        Variable  0
     14    MaxArrayLength                0      11702       Variable  0
```

Disconnect from Server

When you have finished communicating with the server, you should disconnect the client from the server. This is also automatically performed when the client variable goes out of scope in MATLAB.

```
disconnect(uaClient);
```

Read and Write Current OPC UA Server Data

This example shows you how to read and write data to an OPC UA server.

To run this example in your MATLAB® session, you must install and start the Prosys OPC UA Simulation Server. For further information, see the Getting Started section of the Industrial Communication Toolbox™ documentation.

Create a Client and Connect to the Server

You create client objects using the results of a query to the Local Discovery Service using `opcuaserverinfo`, or directly using the host name and port number of the server you are connecting to. In this case, use the host and port number syntax.

```
uaClient = opcua('localhost',53530);
connect(uaClient)
```

Find the DoubleDataItem, FloatDataItem, and Int16DataItem nodes in the StaticData namespace.

```
staticNode = findNodeByName(uaClient.Namespace,'StaticData','-once');
dataItemsNode = findNodeByName(staticNode,'DataItems','-once');
doubleNode = findNodeByName(dataItemsNode,'DoubleDataItem');
floatNode = findNodeByName(dataItemsNode,'FloatDataItem');
int16Node = findNodeByName(dataItemsNode,'Int16DataItem');
nodes = [doubleNode,floatNode,int16Node]
```

```
nodes = 

1x3 OPC UA Node array:
    index        Name       NsInd    Identifier    NodeType  Children
    -----  --------------  -----  --------------  --------  --------
      1    DoubleDataItem  3      DoubleDataItem  Variable  1
      2    FloatDataItem   3      FloatDataItem   Variable  1
      3    Int16DataItem   3      Int16DataItem   Variable  1
```

Read Values from Nodes

Use `readValue` to read the current value of a node. You can query the Value, the Timestamp when the Value was updated, and the Quality associated with the value when written.

```
[v,t,q] = readValue(uaClient,nodes)
```

```
v =

  3×1 cell array

    {[0]}
    {[0]}
    {[0]}


t =
  3×1 datetime array

   19-Mar-2019 02:52:35
   19-Mar-2019 02:52:35
   19-Mar-2019 02:52:35


q = 

OPC UA Quality ID:
    'Good'
    'Good'
    'Good'
```

When you read from multiple nodes, the Values are returned as a cell array. The class of the data on the server is preserved as much as possible.

```
valClasses = cellfun(@class,v,'UniformOutput',false)
```

```
valClasses =

  3×1 cell array

    {'double'}
    {'single'}
    {'int16' }
```

The timestamp is returned as a MATLAB® datetime variable. It represents the time when the source provided the value to the server.

```
t
```

```
t = 

  3×1 datetime array

   19-Mar-2019 02:52:35
   19-Mar-2019 02:52:35
   19-Mar-2019 02:52:35
```

The quality is returned as an OPC UA Quality, which displays as a text description.

```
q
```

```
q = 

OPC UA Quality ID:
    'Good'
    'Good'
    'Good'
```

You can interrogate the Quality to determine characteristics of the returned quality. In this example, the quality is good.

```
isGood(q)
```

```
ans =

  3×1 logical array

   1
   1
   1
```

The value is not interpolated, but is a raw value (stored by the server directly from the sensor).

```
interpolated = isInterpolated(q)

raw = isRaw(q)
```

```
interpolated =

  3×1 logical array

   0
   0
   0


raw =

  3×1 logical array

   1
   1
   1
```

Write Data to Nodes

You can write data to any scalar node. When you write to multiple nodes, you must pass a cell array of values, one for each node to be written.

```
newValues = {12,65,-4};
writeValue(uaClient,nodes,newValues);
```

To verify that the values were written correctly, and retrieve the value again.

```
serverValues = readValue(uaClient,nodes)
```

```
serverValues =

  3×1 cell array

    {[12]}
    {[65]}
    {[-4]}
```

You can update values directly within the cell array and write them back to the server.

```
serverValues{2} = serverValues{2} + 1;
writeValue(uaClient,nodes,serverValues);
```

Read and Write Values With a Single Node

When working with a single node, you receive and can pass and the value directly, without using a cell array.

```
dblValue = readValue(uaClient, doubleNode)
writeValue(uaClient, doubleNode, dblValue+15.6)
newDbl = readValue(uaClient, doubleNode)
```

```
dblValue =

    12


newDbl =

   27.6000
```

Reading and Writing to Nodes Directly

You can write and read directly from the node variable, as long as that node was created from the client (using the Namespace property or browseNamespace) or you passed a client to the `opcuanode` function when creating your node variable.

```
[vals,ts,qual] = readValue(nodes)
writeValue(nodes,v)
```

```
vals =

  3×1 cell array

    {[27.6000]}
    {[     66]}
    {[     -4]}


ts =

  3×1 datetime array

   19-Mar-2019 02:52:36
   19-Mar-2019 02:52:36
   19-Mar-2019 02:52:36


qual =

OPC UA Quality ID:
    'Good'
    'Good'
    'Good'
```

Disconnect from Server

When you have finished communicating with the server, disconnect the client from the server. This is also automatically performed when the client variable goes out of scope in MATLAB.

```
disconnect(uaClient);
```

Read Historical OPC UA Server Data

This example shows you how to read historical data from an OPC UA server.

This example reads data from a Prosys OPC UA Simulation Server v4.0.2. For other Prosys server versions, you might have to modify this code.

To run this example in your MATLAB® session, you must install and start the Prosys OPC UA Simulation Server. For further information, see the Getting Started section of the Industrial Communication Toolbox™ documentation.

Create a Client and Connect to the Server

You create client objects using the results of a query to the Local Discovery Service using `opcuaserverinfo`, or directly using the host name and port number of the server you are connecting to. In this case, connect directly to the OPC UA server on port 53530.

```
uaClient = opcua('localhost',53530);
connect(uaClient);
uaClient.Status
```

```
ans =

    'Connected'
```

Define Nodes to Read Historical Data

The Prosys OPC UA Simulation Server provides simulated signals for nodes in the "Simulation" branch. By default the Simulation Server updates the values each second. Define these nodes using the `opcuanode` function.

```
simNodeIds = {'Random';
              'Triangle';
              'Sinusoid'};
simNodes = opcuanode(3,simNodeIds,uaClient)
```

```
simNodes =

1×3 OPC UA Node array:
    index    Name    NsInd  Identifier  NodeType
    -----  --------  -----  ----------  --------
      1    Random    3      Random      Variable
      2    Triangle  3      Triangle    Variable
      3    Sinusoid  3      Sinusoid    Variable
```

Read Historical Data from Nodes

Use the `readHistory` function to read the history of a node. You must pass a time range in which to read historical data. For the Prosys server, read the most recent 30 seconds of data.

```
dataSample = readHistory(uaClient,simNodes,datetime('now')-seconds(30),datetime('now'))
```

```
dataSample =
1-by-3 OPC UA Data object array:

           Timestamp                   Random                      Triangle                    Si
    -----------------------  -------------------------  -------------------------  ----------
    2019-12-20 01:18:14.000        1.402465 [Good (Raw)]        0.266667 [Good (Raw)]        0.41
    2019-12-20 01:18:15.000        1.044139 [Good (Raw)]        0.000000 [Good (Raw)]        0.00
    2019-12-20 01:18:16.000       -1.857952 [Good (Raw)]       -0.266667 [Good (Raw)]       -0.41
    2019-12-20 01:18:17.000        1.783723 [Good (Raw)]       -0.533333 [Good (Raw)]       -0.81
    2019-12-20 01:18:18.000       -1.095435 [Good (Raw)]       -0.800000 [Good (Raw)]       -1.17
    2019-12-20 01:18:19.000       -1.178567 [Good (Raw)]       -1.066667 [Good (Raw)]       -1.48
    2019-12-20 01:18:20.000       -1.548359 [Good (Raw)]       -1.333333 [Good (Raw)]       -1.73
    2019-12-20 01:18:21.000       -0.438983 [Good (Raw)]       -1.600000 [Good (Raw)]       -1.90
    2019-12-20 01:18:22.000       -0.785842 [Good (Raw)]       -1.866667 [Good (Raw)]       -1.98
    2019-12-20 01:18:23.000        1.419149 [Good (Raw)]       -1.866667 [Good (Raw)]       -1.98
    2019-12-20 01:18:24.000        1.049357 [Good (Raw)]       -1.600000 [Good (Raw)]       -1.90
    2019-12-20 01:18:25.000       -1.932999 [Good (Raw)]       -1.333333 [Good (Raw)]       -1.73
    2019-12-20 01:18:26.000        1.720142 [Good (Raw)]       -1.066667 [Good (Raw)]       -1.48
    2019-12-20 01:18:27.000       -1.170482 [Good (Raw)]       -0.800000 [Good (Raw)]       -1.17
    2019-12-20 01:18:28.000       -1.540274 [Good (Raw)]       -0.533333 [Good (Raw)]       -0.81
    2019-12-20 01:18:29.000       -0.430899 [Good (Raw)]       -0.266667 [Good (Raw)]       -0.41
    2019-12-20 01:18:30.000       -0.869489 [Good (Raw)]       -0.000000 [Good (Raw)]       -0.00
    2019-12-20 01:18:31.000       -1.630916 [Good (Raw)]        0.266667 [Good (Raw)]        0.41
    2019-12-20 01:18:32.000        1.999292 [Good (Raw)]        0.533333 [Good (Raw)]        0.81
    2019-12-20 01:18:33.000       -0.891333 [Good (Raw)]        0.800000 [Good (Raw)]        1.17
    2019-12-20 01:18:34.000       -1.238192 [Good (Raw)]        1.066667 [Good (Raw)]        1.48
    2019-12-20 01:18:35.000       -0.220548 [Good (Raw)]        1.333333 [Good (Raw)]        1.73
    2019-12-20 01:18:36.000       -0.590339 [Good (Raw)]        1.600000 [Good (Raw)]        1.90
    2019-12-20 01:18:37.000        0.519036 [Good (Raw)]        1.866667 [Good (Raw)]        1.98
    2019-12-20 01:18:38.000        0.172177 [Good (Raw)]        1.866667 [Good (Raw)]        1.98
    2019-12-20 01:18:39.000       -0.589250 [Good (Raw)]        1.600000 [Good (Raw)]        1.90
    2019-12-20 01:18:40.000       -0.959042 [Good (Raw)]        1.333333 [Good (Raw)]        1.73
    2019-12-20 01:18:41.000        0.425527 [Good (Raw)]        1.066667 [Good (Raw)]        1.48
    2019-12-20 01:18:42.000        0.078668 [Good (Raw)]        0.800000 [Good (Raw)]        1.17
    2019-12-20 01:18:43.000        1.188043 [Good (Raw)]        0.533333 [Good (Raw)]        0.81
```

Read Historical Data at Specific Times

You can ask the server to retrieve data at specific times. If the server does not have an archived value for that specific time, an interpolated (or extrapolated) value is returned. Use the `readAtTime` function to retrieve data each minute for the last 10 minutes.

```
timesToReturn = datetime('now')-minutes(10):minutes(1):datetime('now');
dataRegular = readAtTime(uaClient,simNodes,timesToReturn)
```

```
dataRegular = 

1-by-3 OPC UA Data object array:

           Timestamp                   Random                      Triangle                    Si
    -----------------------  -------------------------  -------------------------  ----------
    2019-12-20 01:08:44.000       -0.083361 [Good (Raw)]        0.266667 [Good (Raw)]        0.41
    2019-12-20 01:09:44.000        0.043744 [Good (Raw)]        0.266667 [Good (Raw)]        0.41
    2019-12-20 01:10:44.000        1.199272 [Good (Raw)]        0.266667 [Good (Raw)]        0.41
    2019-12-20 01:11:44.000        1.259184 [Good (Raw)]        0.266667 [Good (Raw)]        0.41
    2019-12-20 01:12:44.000        0.193783 [Good (Raw)]        0.266667 [Good (Raw)]        0.41
    2019-12-20 01:13:44.000       -1.585967 [Good (Raw)]        0.266667 [Good (Raw)]        0.4
    2019-12-20 01:14:44.000        1.073438 [Good (Raw)]        0.266667 [Good (Raw)]        0.4
    2019-12-20 01:15:44.000        0.099768 [Good (Raw)]        0.266667 [Good (Raw)]        0.4
    2019-12-20 01:16:44.000       -1.368735 [Good (Raw)]        0.266667 [Good (Raw)]        0.4
    2019-12-20 01:17:44.000        1.791922 [Good (Raw)]        0.266667 [Good (Raw)]        0.4
    2019-12-20 01:18:44.000        0.818252 [Good (Raw)]        0.266667 [Good (Raw)]        0.4
```

Read Processed Data from the Server

OPC UA Servers provide aggregate functions for returning preprocessed data to clients. This is most useful when you need to query data over a large period of time.

Query the `AggregateFunctions` property of a connected client to find out what aggregate functions the server supports.

```
uaClient.AggregateFunctions
```

```
ans =

  14×1 cell array

    {'Interpolative'    }
    {'Average'          }
    {'Minimum'          }
    {'Maximum'          }
    {'MinimumActualTime'}
    {'MaximumActualTime'}
    {'Range'            }
    {'Count'            }
    {'Start'            }
    {'End'              }
    {'Delta'            }
    {'WorstQuality'     }
    {'StartBound'       }
    {'EndBound'         }
```

Read the Average value for each 30 second period over the last 10 minutes.

```
dataAverage = readProcessed(uaClient,simNodes,'Average',seconds(30),datetime('now')-minutes(10),d
```

```
dataAverage = 

1-by-3 OPC UA Data object array:

           Timestamp                      Random                             Triangle
    -----------------------  --------------------------------  ----------------------------------
    2019-12-20 01:08:44.000      -0.008396 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:09:14.000       0.071422 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:09:44.000       0.034084 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:10:14.000       0.190256 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:10:44.000       0.088148 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:11:14.000       0.065122 [Good (Calculated)]       0.000000 [Good (Calculated
    2019-12-20 01:11:44.000      -0.057444 [Good (Calculated)]       0.000000 [Good (Calculated
    2019-12-20 01:12:14.000      -0.047782 [Good (Calculated)]       0.000000 [Good (Calculated
    2019-12-20 01:12:44.000       0.253328 [Good (Calculated)]       0.000000 [Good (Calculated
    2019-12-20 01:13:14.000      -0.018746 [Good (Calculated)]       0.000000 [Good (Calculated
    2019-12-20 01:13:44.000       0.103775 [Good (Calculated)]       0.000000 [Good (Calculated
    2019-12-20 01:14:14.000       0.010857 [Good (Calculated)]       0.000000 [Good (Calculated
    2019-12-20 01:14:44.000      -0.370672 [Good (Calculated)]       0.000000 [Good (Calculated
    2019-12-20 01:15:14.000      -0.198687 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:15:44.000      -0.025481 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:16:14.000       0.067565 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:16:44.000       0.085904 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:17:14.000       0.018061 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:17:44.000      -0.033414 [Good (Calculated)]      -0.000000 [Good (Calculated
    2019-12-20 01:18:14.000      -0.205573 [Good (Calculated)]      -0.000000 [Good (Calculated
```

Read the Average value for each half second period over the last 5 seconds. Note how the quality of the data includes Good quality, and Bad quality where there is no data available to perform the calculation.

```
dataMixedQuality = readProcessed(uaClient,simNodes,'Average',seconds(0.5),datetime('now')-seconds
```

```
dataMixedQuality = 

1-by-3 OPC UA Data object array:

           Timestamp                     Random                             Triangle
    -----------------------  --------------------------------  --------------------------------
    2019-12-20 01:18:39.000       0.000000 [Bad:NoData (Raw)]       0.000000 [Bad:NoData (Raw)
    2019-12-20 01:18:39.500      -0.589250 [Good (Calculated)]       1.600000 [Good (Calculated
    2019-12-20 01:18:40.000       0.000000 [Bad:NoData (Raw)]       0.000000 [Bad:NoData (Raw)
    2019-12-20 01:18:40.500      -0.959042 [Good (Calculated)]       1.333333 [Good (Calculated
    2019-12-20 01:18:41.000       0.000000 [Bad:NoData (Raw)]       0.000000 [Bad:NoData (Raw)
    2019-12-20 01:18:41.500       0.425527 [Good (Calculated)]       1.066667 [Good (Calculated
    2019-12-20 01:18:42.000       0.000000 [Bad:NoData (Raw)]       0.000000 [Bad:NoData (Raw)
    2019-12-20 01:18:42.500       0.078668 [Good (Calculated)]       0.800000 [Good (Calculated
    2019-12-20 01:18:43.000       0.000000 [Bad:NoData (Raw)]       0.000000 [Bad:NoData (Raw)
    2019-12-20 01:18:43.500       1.188043 [Good (Calculated)]       0.533333 [Good (Calculated
```

Filter the quality of the data to return only the Good data.

```
dataGood = filterByQuality(dataMixedQuality,'good')
```

```
dataGood = 

1-by-3 OPC UA Data object array:

           Timestamp                     Random                             Triangle
    -----------------------  --------------------------------  --------------------------------
    2019-12-20 01:18:39.500      -0.589250 [Good (Calculated)]       1.600000 [Good (Calculated
    2019-12-20 01:18:40.500      -0.959042 [Good (Calculated)]       1.333333 [Good (Calculated
    2019-12-20 01:18:41.500       0.425527 [Good (Calculated)]       1.066667 [Good (Calculated
    2019-12-20 01:18:42.500       0.078668 [Good (Calculated)]       0.800000 [Good (Calculated
    2019-12-20 01:18:43.500       1.188043 [Good (Calculated)]       0.533333 [Good (Calculated
```

Disconnect from Server

When you have finished communicating with the server, disconnect the client from the server. This is also automatically performed when the client variable goes out of scope in MATLAB®.

```
disconnect(uaClient);
```

Visualize and Preprocess OPC UA Data

This example shows you how to work with OPC UA Data objects.

You create OPC UA Data objects when you read historical data from an OPC UA server. OPC UA Data objects allow you to store, visualize and manipulate historical data before converting that data to builtin data types for further processing in MATLAB.

For more information on generating OPC UA Data objects, see the example “Read Historical OPC UA Server Data” on page 21-56.

Load Sample OPC UA Data Set

Load the sample data into the workspace.

```
load demoUA_SampleData
```

Display OPC UA Data objects

Examine the workspace to see what variables have been loaded.

```
whos
```

```
  Name            Size            Bytes  Class         Attributes

  dataSample      1x3              5926  opc.ua.Data
```

Display a summary of the sample data.

```
summary(dataSample)
```

```
1-by-3 OPC UA Data object:

     Name        Value            Start Timestamp          End Timestamp            Quality
    ------  ----------------  -----------------------  -----------------------  ----------------
    Double  9 double values   2015-04-22 09:00:10.000  2015-04-22 09:01:30.000  3 unique qualitie
    Float   12 single values  2015-04-22 09:00:02.000  2015-04-22 09:01:30.000  3 unique qualitie
    Int32   12 int32 values   2015-04-22 09:00:02.000  2015-04-22 09:01:30.000  3 unique qualitie
```

The data object contains three data sets. The first element `Double` contains 9 values, the second and third have 12 values each.

See whether the `Float` and `Int32` data sets have the same timestamp.

```
arrayHasSameTimestamp(dataSample(2:3))
```

```
ans =

   1
```

Display the `Float` and `Int32` data sets together. Because all the elements have the same timestamp, a table of values can be displayed

```
dataSample(2:3)
```

```
ans = 

1-by-2 OPC UA Data object array:

          Timestamp                      Float                             Int32
    -----------------------  ------------------------------  -------------------------------
    2015-04-22 09:00:02.000       10.000000 [Good (Raw)]                    10 [Good (Raw)]
    2015-04-22 09:00:25.000       20.000000 [Good (Raw)]                    20 [Good (Raw)]
    2015-04-22 09:00:28.000       25.000000 [Good (Raw)]                    25 [Good (Raw)]
    2015-04-22 09:00:40.000       30.000000 [Good (Raw)]                    30 [Good (Raw)]
    2015-04-22 09:00:42.000        0.000000 [Bad (Raw)]                      0 [Bad (Raw)]
    2015-04-22 09:00:48.000        4.000000 [Good (Raw)]                    40 [Good (Raw)]
    2015-04-22 09:00:52.000       50.000000 [Good (Raw)]                    50 [Good (Raw)]
    2015-04-22 09:01:12.000       60.000000 [Good (Raw)]                    60 [Good (Raw)]
    2015-04-22 09:01:17.000       70.000000 [Uncertain (Raw)]               70 [Uncertain (Raw)]
    2015-04-22 09:01:23.000       70.000000 [Good (Raw)]                    70 [Good (Raw)]
    2015-04-22 09:01:26.000       80.000000 [Good (Raw)]                    80 [Good (Raw)]
    2015-04-22 09:01:30.000       90.000000 [Good (Raw)]                    90 [Good (Raw)]
```

Change the Date Display Format

Get the current date display format using `opc.getDateDisplayFormat`.

```
origFormat = opc.getDateDisplayFormat;
```

Change the display format to standard US date format and display the value again.

```
opc.setDateDisplayFormat('mm/dd/yyyy HH:MM AM');
dataSample(2:3)
```

```
ans = 

1-by-2 OPC UA Data object array:

        Timestamp                      Float                             Int32
    -------------------  ------------------------------  -------------------------------
    04/22/2015  9:00 AM       10.000000 [Good (Raw)]                    10 [Good (Raw)]
    04/22/2015  9:00 AM       20.000000 [Good (Raw)]                    20 [Good (Raw)]
    04/22/2015  9:00 AM       25.000000 [Good (Raw)]                    25 [Good (Raw)]
    04/22/2015  9:00 AM       30.000000 [Good (Raw)]                    30 [Good (Raw)]
    04/22/2015  9:00 AM        0.000000 [Bad (Raw)]                      0 [Bad (Raw)]
    04/22/2015  9:00 AM        4.000000 [Good (Raw)]                    40 [Good (Raw)]
    04/22/2015  9:00 AM       50.000000 [Good (Raw)]                    50 [Good (Raw)]
    04/22/2015  9:01 AM       60.000000 [Good (Raw)]                    60 [Good (Raw)]
    04/22/2015  9:01 AM       70.000000 [Uncertain (Raw)]               70 [Uncertain (Raw)]
    04/22/2015  9:01 AM       70.000000 [Good (Raw)]                    70 [Good (Raw)]
    04/22/2015  9:01 AM       80.000000 [Good (Raw)]                    80 [Good (Raw)]
    04/22/2015  9:01 AM       90.000000 [Good (Raw)]                    90 [Good (Raw)]
```

Reset the display format to the default.

```
opc.setDateDisplayFormat('default')
```

```
ans = 
yyyy-mm-dd HH:MM:SS.FFF
```

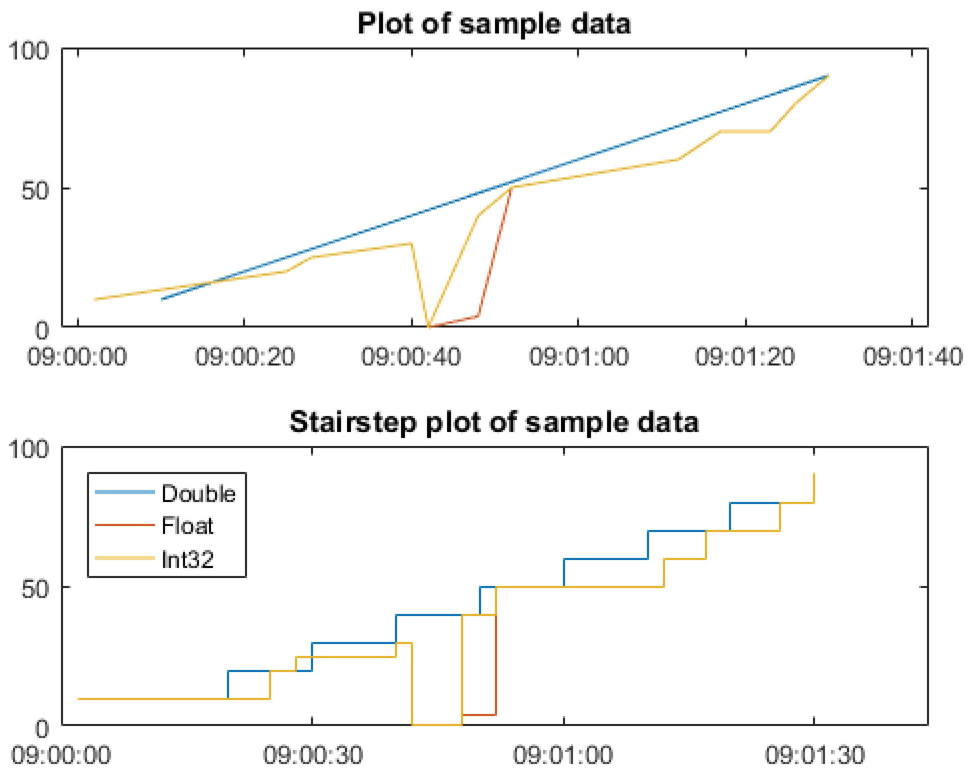
Reset the display format to the original value.

```
opc.setDateDisplayFormat(origFormat);
```

Visualize OPC UA Data

Visualize OPC UA Data using the `plot` and `stairs` functions on the data object.

```
axH1 = subplot(2,1,1);
plot(dataSample);
title('Plot of sample data');
axH2 = subplot(2,1,2);
stairs(dataSample);
title('Stairstep plot of sample data');
legend('Location', 'NorthWest')
```



Resample OPC UA Data

The data in the `dataSample` set has different timestamps.

```
arrayHasSameTimestamp(dataSample)
```

```
ans =
    0
```

Attempt to convert the data to a double array. The conversion will fail.

```
try
    vals = double(dataSample);
catch exc
    disp(exc.message)
end
```

```
Conversion to double failed. All elements of the OPC Data object must have the same time stamp.
Consider using 'TSUNION', 'TSINTERSECT' or 'RESAMPLE' on the Data object.
```

The intersection of the data timestamps results in a smaller data set containing the common timestamps from all elements.

```
dataIntersect = tsintersect(dataSample)
```

```
dataIntersect = 

1-by-3 OPC UA Data object array:

           Timestamp                    Double                     Float                    I
    -----------------------  -------------------------  -------------------------  -----------
    2015-04-22 09:00:40.000       40.000000 [Bad (Raw)]       30.000000 [Good (Raw)]
    2015-04-22 09:01:30.000       90.000000 [Good (Raw)]      90.000000 [Good (Raw)]
```

Convert the data object into a double array.

```
vals = double(dataIntersect)
```

```
vals =

    40    30    30
    90    90    90
```

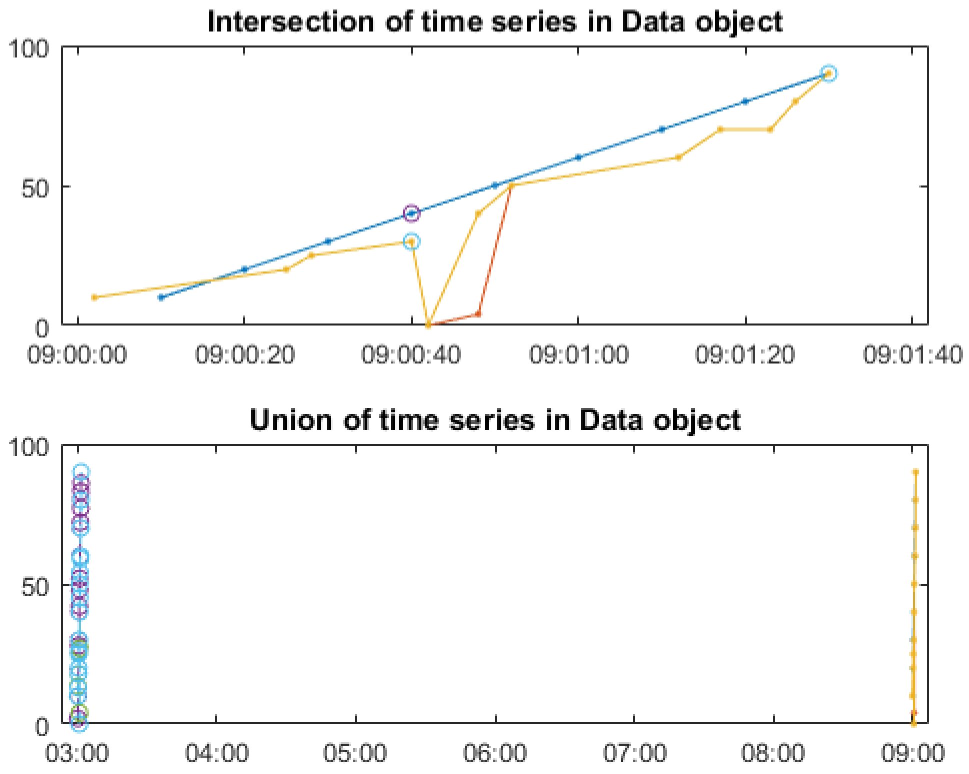
Use `tsunion` to return the union of time series in a Data object. New values are interpolated using the method supplied (or linear interpolation if no method is supplied). The quality is set to "Interpolated" for those new values.

```
dataUnion = tsunion(dataSample)
```

```
dataUnion = 

1-by-3 OPC UA Data object array:

           Timestamp                               Double                                    
    -----------------------  ---------------------------------------------  ---------------
    2015-04-22 03:00:02.000        2.000000 [Uncertain:Subnormal (Interpolated)]      10.000000
    2015-04-22 03:00:10.000       10.000000 [Good (Raw)]                              13.478261
    2015-04-22 03:00:20.000       20.000000 [Good (Raw)]                              17.826086
    2015-04-22 03:00:25.000       25.000000 [Good (Interpolated)]                     20.000000
    2015-04-22 03:00:28.000       28.000000 [Good (Interpolated)]                            25.000000
    2015-04-22 03:00:30.000       30.000000 [Good (Raw)]                                     25.833334
    2015-04-22 03:00:40.000       40.000000 [Bad (Raw)]                                      30.000000
    2015-04-22 03:00:42.000       42.000000 [Good (Interpolated)]                             0.000000
    2015-04-22 03:00:48.000       48.000000 [Good (Interpolated)]                             4.000000
    2015-04-22 03:00:50.000       50.000000 [Good (Raw)]                                     27.000000
    2015-04-22 03:00:52.000       52.000000 [Good (Interpolated)]                            50.000000
    2015-04-22 03:01:00.000       60.000000 [Good (Raw)]                                     54.000000
    2015-04-22 03:01:10.000       70.000000 [Uncertain (Raw)]                                59.000000
    2015-04-22 03:01:12.000       72.000000 [Good (Interpolated)]                            60.000000
    2015-04-22 03:01:17.000       77.000000 [Good (Interpolated)]                            70.000000
    2015-04-22 03:01:20.000       80.000000 [Good (Raw)]                                     70.000000
    2015-04-22 03:01:23.000       83.000000 [Good (Interpolated)]                            70.000000
    2015-04-22 03:01:26.000       86.000000 [Good (Interpolated)]                            80.000000
    2015-04-22 03:01:30.000       90.000000 [Good (Raw)]                                     90.000000
```

Plot the data with markers to show how the methods work.

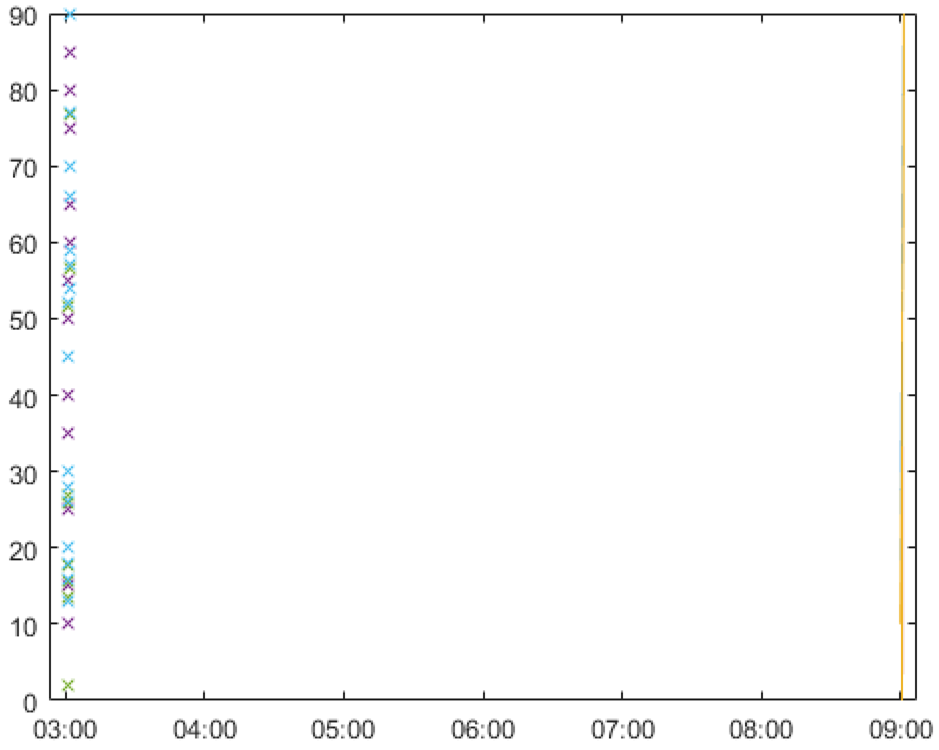
```
subplot(2,1,1);
plot(dataSample, 'Marker','.');
hold all
plot(dataIntersect, 'Marker','o', 'LineStyle','none');
title('Intersection of time series in Data object');
subplot(2,1,2);
plot(dataSample, 'Marker','.');
hold all
plot(dataUnion, 'Marker','o', 'LineStyle','--');
title('Union of time series in Data object');
```



Resample the small data set at specified time steps.

```
newTS = dataSample(1).Timestamp(1):seconds(5):dataSample(1).Timestamp(end);
dataResampled = resample(dataSample,newTS)
figure;
plot(dataSample);
hold all
plot(dataResampled, 'Marker','x', 'Linestyle','none');
```

```
dataResampled =

1-by-3 OPC UA Data object array:

           Timestamp                           Double                                   Float
    -----------------------  ----------------------------------  -------------------------------
    2015-04-22 03:00:10.000      10.000000 [Good (Raw)]                13.478261 [Good (Interpo
    2015-04-22 03:00:15.000      15.000000 [Good (Interpolated)]       15.652174 [Good (Interpo
    2015-04-22 03:00:20.000      20.000000 [Good (Raw)]                17.826086 [Good (Interpo
    2015-04-22 03:00:25.000      25.000000 [Good (Interpolated)]       20.000000 [Good (Raw)]
    2015-04-22 03:00:30.000      30.000000 [Good (Raw)]                25.833334 [Good (Interpo
    2015-04-22 03:00:35.000      35.000000 [Good (Interpolated)]       27.916666 [Good (Interpo
    2015-04-22 03:00:40.000      40.000000 [Bad (Raw)]                 30.000000 [Good (Raw)]
    2015-04-22 03:00:45.000      45.000000 [Good (Interpolated)]        2.000000 [Good (Interpo
    2015-04-22 03:00:50.000      50.000000 [Good (Raw)]                27.000000 [Good (Interpo
    2015-04-22 03:00:55.000      55.000000 [Good (Interpolated)]       51.500000 [Good (Interpo
    2015-04-22 03:01:00.000      60.000000 [Good (Raw)]                54.000000 [Good (Interpo
    2015-04-22 03:01:05.000      65.000000 [Good (Interpolated)]       56.500000 [Good (Interpo
    2015-04-22 03:01:10.000       70.000000 [Uncertain (Raw)]        59.000000 [Good (Interpo
    2015-04-22 03:01:15.000       75.000000 [Good (Interpolated)]    66.000000 [Good (Interpo
    2015-04-22 03:01:20.000       80.000000 [Good (Raw)]             70.000000 [Good (Interpo
    2015-04-22 03:01:25.000       85.000000 [Good (Interpolated)]    76.666664 [Good (Interpo
    2015-04-22 03:01:30.000       90.000000 [Good (Raw)]             90.000000 [Good (Raw)]
```



Filter Data by Quality

Find only the Good data from the second element of resampled data set

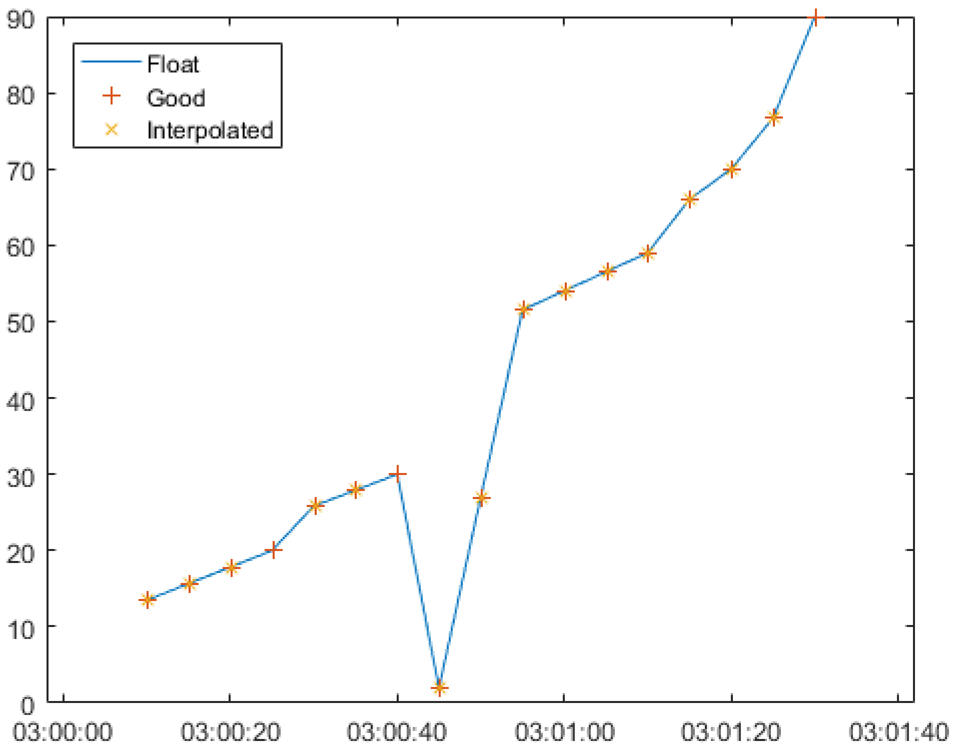
```
resampledGood = filterByQuality(dataResampled(2), 'good')
```

```
resampledGood =

1-by-1 OPC UA Data object array:

           Timestamp                         Float
    -----------------------  ----------------------------------
    2015-04-22 03:00:10.000       13.478261 [Good (Interpolated)]
    2015-04-22 03:00:15.000       15.652174 [Good (Interpolated)]
    2015-04-22 03:00:20.000       17.826086 [Good (Interpolated)]
    2015-04-22 03:00:25.000       20.000000 [Good (Raw)]
    2015-04-22 03:00:30.000       25.833334 [Good (Interpolated)]
    2015-04-22 03:00:35.000       27.916666 [Good (Interpolated)]
    2015-04-22 03:00:40.000       30.000000 [Good (Raw)]
    2015-04-22 03:00:45.000        2.000000 [Good (Interpolated)]
    2015-04-22 03:00:50.000       27.000000 [Good (Interpolated)]
    2015-04-22 03:00:55.000       51.500000 [Good (Interpolated)]
    2015-04-22 03:01:00.000       54.000000 [Good (Interpolated)]
    2015-04-22 03:01:05.000       56.500000 [Good (Interpolated)]
    2015-04-22 03:01:10.000       59.000000 [Good (Interpolated)]
    2015-04-22 03:01:15.000       66.000000 [Good (Interpolated)]
    2015-04-22 03:01:20.000       70.000000 [Good (Interpolated)]
    2015-04-22 03:01:25.000       76.666664 [Good (Interpolated)]
    2015-04-22 03:01:30.000       90.000000 [Good (Raw)]
```

Filter the second element of the resampled data to return only the Interpolated data. Visualize the filtered data with the original.

```
resampledInterpolated = filterByQuality(dataResampled(2), 'Origin','interpolated')

figure;
plot(dataResampled(2))
hold on
plot(resampledGood, 'Marker', '+', 'Linestyle','none', 'DisplayName', 'Good');
plot(resampledInterpolated, 'Marker','x', 'Linestyle','none', 'DisplayName', 'Interpolated');
legend('Location', 'NorthWest')
```

```
resampledInterpolated = 

1-by-1 OPC UA Data object array:

           Timestamp                         Float                
    -----------------------  ----------------------------------
    2015-04-22 03:00:10.000       13.478261 [Good (Interpolated)]
    2015-04-22 03:00:15.000       15.652174 [Good (Interpolated)]
    2015-04-22 03:00:20.000       17.826086 [Good (Interpolated)]
    2015-04-22 03:00:30.000       25.833334 [Good (Interpolated)]
    2015-04-22 03:00:35.000       27.916666 [Good (Interpolated)]
    2015-04-22 03:00:45.000        2.000000 [Good (Interpolated)]
    2015-04-22 03:00:50.000       27.000000 [Good (Interpolated)]
    2015-04-22 03:00:55.000       51.500000 [Good (Interpolated)]
    2015-04-22 03:01:00.000       54.000000 [Good (Interpolated)]
    2015-04-22 03:01:05.000       56.500000 [Good (Interpolated)]
    2015-04-22 03:01:10.000       59.000000 [Good (Interpolated)]
    2015-04-22 03:01:15.000       66.000000 [Good (Interpolated)]
    2015-04-22 03:01:20.000       70.000000 [Good (Interpolated)]
    2015-04-22 03:01:25.000       76.666664 [Good (Interpolated)]
```


Float
Good
Interpolated
0
10
20
30
40
50
60
70
80
90
03:00:00
03:00:20
03:00:40
03:01:00
03:01:20
03:01:40

Read and Write to an OPC Data Access Server from Simulink

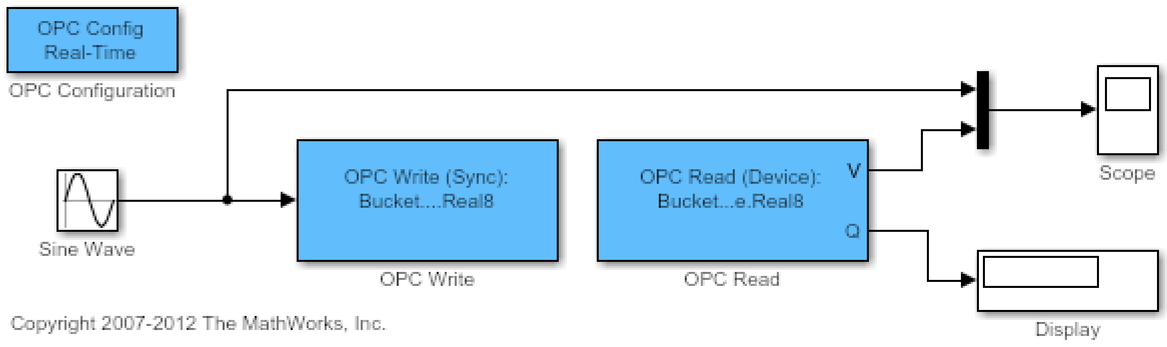
This example shows you how to exchange data between Simulink and OPC Data Access servers.

PREREQUISITES:

- "Install a Simulation Server for OPC Examples" on page 21-2

Model Description

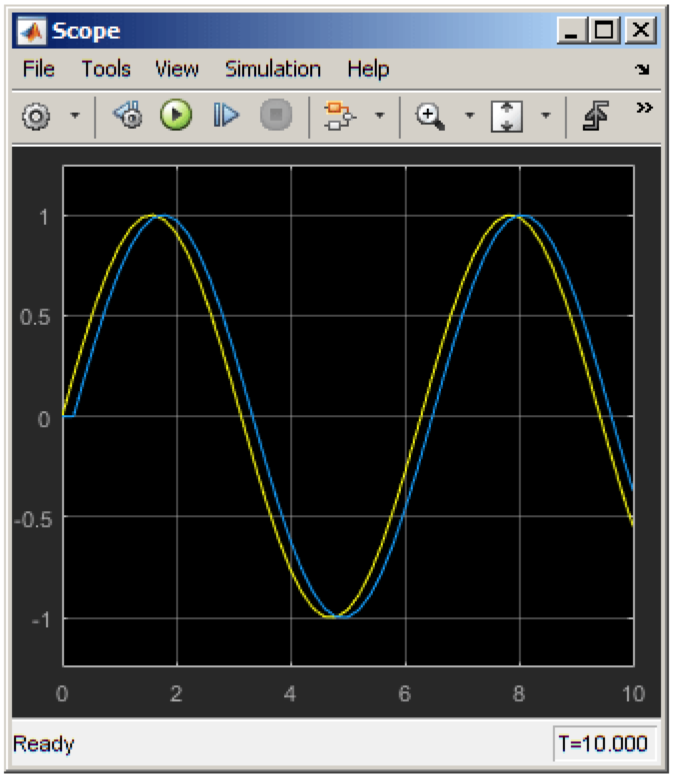
In the following model, all OPC blocks are highlighted in blue.



The `OPC Config` block defines the servers to use in the model, the pseudo-realtime behaviour of the model when it is simulated, and the actions to take when OPC-specific events occur (such as a pseudo-realtime violation, server shutdown, etc.)

The signal from the `Sine Wave` block is written to the OPC Server using the `OPC Write` block. The same signal is read back from the server using the `OPC Read` block, and displayed in the `Scope` together with the original Sine Wave signal. The OPC data quality is shown in the `Display` block.

Understanding the Simulation Results



The Scope shows that the Sine Wave signal is delayed by only one sample. The `OPC Write` block's priority is set higher than the `OPC Read` block to ensure that the read operation occurs after the write operation. This ensures only one sample delay between the write and read operation.

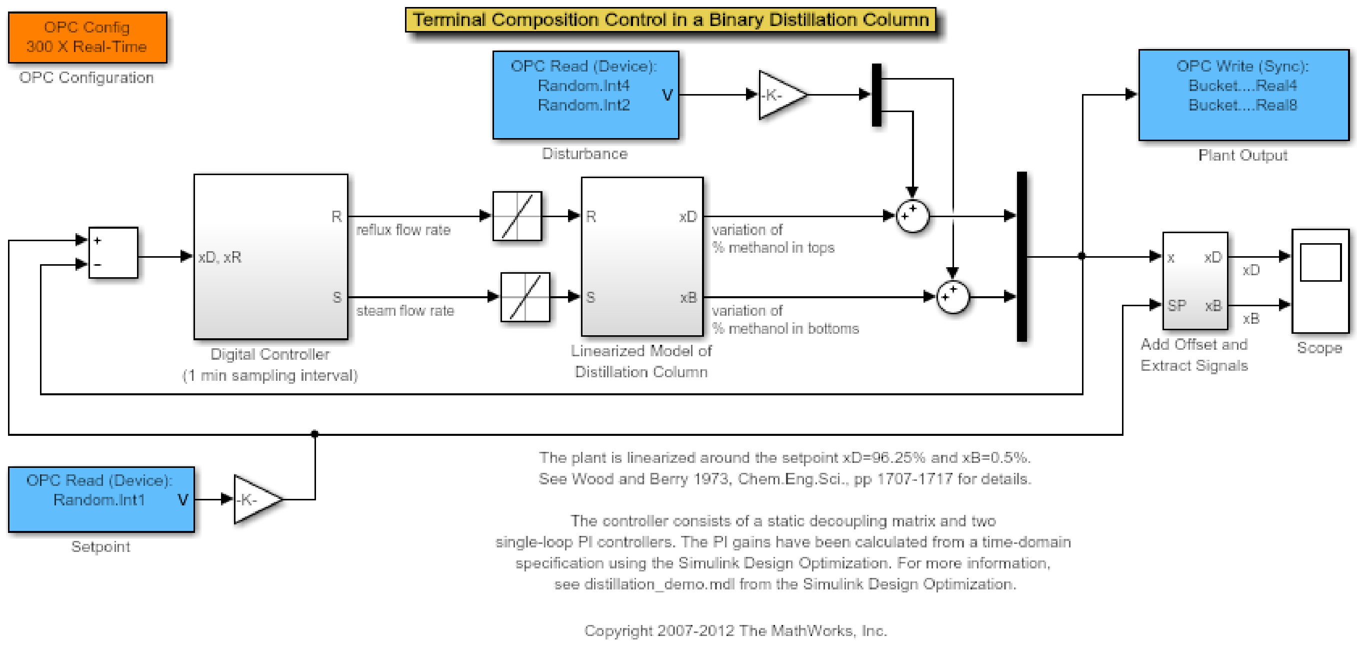
Use OPC Data to Test a Binary Distillation Column Plant Model

This example shows how to use data from an OPC server to test composition control of a binary distillation column model.

PREREQUISITES:

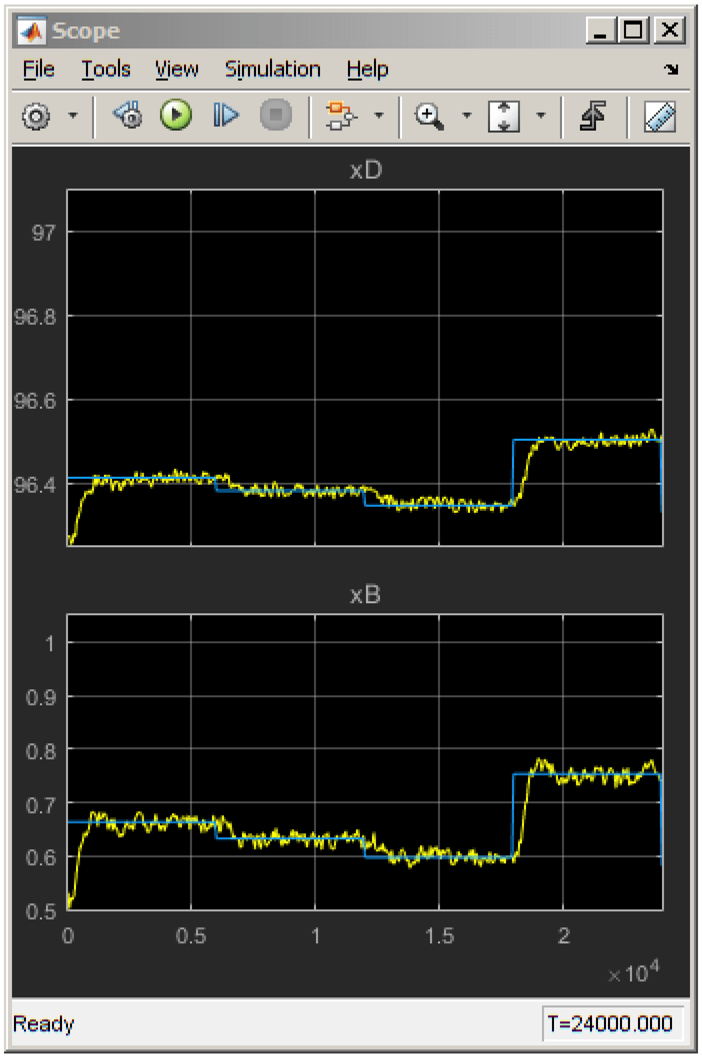
- "Install a Simulation Server for OPC Examples" on page 21-2

Model Description


Terminal Composition Control in a Binary Distillation Column
OPC Config
300 X Real-Time
OPC Configuration
OPC Read (Device):
Random.Int4
Random.Int2
V
Disturbance
-K-
OPC Write (Sync):
Bucket....Real4
Bucket....Real8
Plant Output
xD, xR
R
S
reflux flow rate
steam flow rate
Digital Controller
(1 min sampling interval)
R
S
xD
xB
Linearized Model of
Distillation Column
variation of
% methanol in tops
variation of
% methanol in bottoms
x
SP
xD
xB
Add Offset and
Extract Signals
Scope
OPC Read (Device):
Random.Int1
V
-K-
Setpoint
The plant is linearized around the setpoint xD=96.25% and xB=0.5%.
See Wood and Berry 1973, Chem.Eng.Sci., pp 1707-1717 for details.
The controller consists of a static decoupling matrix and two
single-loop PI controllers. The PI gains have been calculated from a time-domain
specification using the Simulink Design Optimization. For more information,
see distillation_demo.mdl from the Simulink Design Optimization.
Copyright 2007-2012 The MathWorks, Inc.

The distillation column controller has been tuned using Simulink® Design Optimization™. For more information, see `distillation_demo.mdl` from Simulink Design Optimization.

The OPC server provides a random signal for the set-points, and for injecting an output disturbance on the process (see the `OPC Read` blocks, colored blue). The plant outputs are written back to the OPC server using the blue `OPC Write` block.



Running Simulink Models Faster Than Real-Time

Note how in this example runs the model at 300 times real-time, using the speedup factor in the `OPC Configuration` block (colored orange). The distillation column model has very slow time constants (on the order of 20 minutes), and it is desirable to test the simulation at faster rates than real-time. By using the speedup factor, you can simulate 400 minutes in approximately 80 seconds. This model could be connected to an HMI or SCADA system to train operators in expected plant responses, or to validate the SCADA system against desired performance specifications.

Get Started Accessing Data from a PI Server

This example shows you how to connect to an OSIsoft™ PI Server and locate asset information stored in its data archive. Running this example requires that an OSIsoft PI System is installed. The demo tags used in this example were provided by OSIsoft and may be downloaded from the following location:

https://learning.osisoft.com/asset-based-af-example-kits

The PI Server is capable of storing decades of real-time data from hundreds of assets. The MATLAB® interface for the PI System leverages the system's Asset Framework (AF) to access time series data of your assets.

Create Client and Connect to Server

Connect to a PI Server using the `piclient` function. In this example the Windows computer name is used as the PI AF Server name. Your situation may vary depending on PI System configuration.

```
host = getenv("COMPUTERNAME");
client = piclient(host);
```

List All Tags

Create a list of all tags available on the PI Server using the `tags` method. Depending on your system, this query may return a large amount of data. If you have an extensive list of tags that makes this too slow or impractical, you may want to skip this step.

A tag is used by the PI System as an alias or shortcut to represent an asset attribute such as voltage, current, temperature, etc. Some tag names are short, others may be long and descriptive or include a unique ID.

```
allTags = tags(client)
```

```
allTags=478×1 table
                                                              Tags
    ______________________________________________________________________________________________________

    "Flynn I.Active Power Generated.1b86ebf3-1c0a-52bd-3222-38e6660052f2"
    "Flynn I.Hydro Unit Attention Percentage.4ae999f5-f5ae-535c-3f43-fcb3775ee8a4"
    "Flynn I.Hydro Unit Condition.f3ec518f-1059-5c79-00cf-28c97d06714b"
    "Flynn II.Active Power Generated.1b86eb8e-1c0a-52bd-3222-38e6660052f2"
    "Flynn II.Hydro Unit Attention Percentage.4ae99988-f5ae-535c-3f43-fcb3775ee8a4"
    "Flynn II.Hydro Unit Condition.f3ec51f2-1059-5c79-00cf-28c97d06714b"
    "Flynn River Hydro.Actual Power Generated.1670a44f-b666-5758-3021-6f6f8a37127d"
    "Flynn River Hydro.Hydro Attention Percentage.e7b2c9f9-ef29-54be-0f3a-41a791e7bfd1"
    "Flynn River Hydro.Hydro Condition.1bf06b5f-9e31-5615-1701-19b9b333770f"
    "GU1 Generator.Analysis Generator Condition - Hours Since Last Maintenance.d7df31d2-dbb3-5ff
    "GU1 Generator.Analysis Generator Cooling - Bearing Temperature.7c20f57e-0f25-5474-3d73-b839
    "GU1 Generator.Analysis Generator Cooling - Cooling Water Output Temperature.844da429-3a04-5
    "GU1 Generator.Analysis Generator Cooling - Cooling Water Pressure.e217c9ab-825a-5e3b-1915-6
    "GU1 Generator.Analysis Generator Cooling - Core Temperature.204ab467-880e-5691-189c-3bc20e8
    "GU1 Generator.Analysis Generator Cooling - Rotor Winding Temperature.1c7fa6fd-9eaf-5fea-30c
    "GU1 Generator.Analysis Generator Lubricating -  Lubricant Oil Output Temperature.d6db886f-e
      ⋮
```

Narrow List of Tags to Specific Facility Using Wildcard

You could choose to browse the list of tags returned in the previous step to locate a particular tag and assign it to a new variable. But it is more convenient to narrow the search using additional input parameters in the `tags` method. For example, based on the list of tags returned in the previous step, specify a prefix in your tag search to narrow the results. The following example uses a wildcard to find all tags with the prefix "OSIDemo_Flynn I.".

```
tagsFlynnI = tags(client, Name = "OSIDemo_Flynn I.*")
```

```
tagsFlynnI=7×1 table
                                      Tags
    ________________________________________________________________________

    "OSIDemo_Flynn I.Dam Gates Opening.b2156e0a-0b4d-555c-3edf-cea1339a96b7"
    "OSIDemo_Flynn I.Penstock Flow.322006e7-2097-5f17-0e4d-1be282f8c36c"
    "OSIDemo_Flynn I.Penstock Opening.d49c75c6-5ef9-57a8-0366-9d71e9b595cf"
    "OSIDemo_Flynn I.Penstock Pressure.c550b9de-f101-53be-1b5a-84e0f3c22991"
    "OSIDemo_Flynn I.Reservoir Level.1a4f7e16-c4ad-55ea-0731-ca5ba86bcaf7"
    "OSIDemo_Flynn I.Water PH.42e7dc15-656a-5aa7-181c-6adee980d38a"
    "OSIDemo_Flynn I.Water Temperature.18d94851-7252-5c87-1fb3-d769fb949f15"
```

Use Multiple Wildcards to Narrow List of Tags

You may also use multiple wildcards to narrow a list of tags. Notice the example below uses a wildcard before and after "Demo". This returns all tags containing the string "Demo". Notice the results now include tags containing both "Flynn I" and "Flynn II".

```
tagsDemo = tags(client, Name = "*Demo*")
```

```
tagsDemo=194×1 table
                                        Tags
    ____________________________________________________________________________

    "OSIDemo_Flynn I.Dam Gates Opening.b2156e0a-0b4d-555c-3edf-cea1339a96b7"
    "OSIDemo_Flynn I.Penstock Flow.322006e7-2097-5f17-0e4d-1be282f8c36c"
    "OSIDemo_Flynn I.Penstock Opening.d49c75c6-5ef9-57a8-0366-9d71e9b595cf"
    "OSIDemo_Flynn I.Penstock Pressure.c550b9de-f101-53be-1b5a-84e0f3c22991"
    "OSIDemo_Flynn I.Reservoir Level.1a4f7e16-c4ad-55ea-0731-ca5ba86bcaf7"
    "OSIDemo_Flynn I.Water PH.42e7dc15-656a-5aa7-181c-6adee980d38a"
    "OSIDemo_Flynn I.Water Temperature.18d94851-7252-5c87-1fb3-d769fb949f15"
    "OSIDemo_Flynn II.Dam Gates Opening.b2156e77-0b4d-555c-3edf-cea1339a96b7"
    "OSIDemo_Flynn II.Penstock Flow.3220069a-2097-5f17-0e4d-1be282f8c36c"
    "OSIDemo_Flynn II.Penstock Opening.d49c75bb-5ef9-57a8-0366-9d71e9b595cf"
    "OSIDemo_Flynn II.Penstock Pressure.c550b9a3-f101-53be-1b5a-84e0f3c22991"
    "OSIDemo_Flynn II.Reservoir Level.1a4f7e6b-c4ad-55ea-0731-ca5ba86bcaf7"
    "OSIDemo_Flynn II.Water PH.42e7dc68-656a-5aa7-181c-6adee980d38a"
    "OSIDemo_Flynn II.Water Temperature.18d9482c-7252-5c87-1fb3-d769fb949f15"
    "OSIDemo_GU1 Generator.Active Power.4db83f0a-ff87-5c67-385a-83cfe3ac560d"
    "OSIDemo_GU1 Generator.Axial Vibration.373a6144-442d-5e69-1079-5986bf866fa1"
      ⋮
```

Narrow List of Tags Using MATLAB Pattern Matching

Narrow the list of tags using the `contains` function. This example makes use of the `tagsDemo` list from a previous step, returning all tags containing the string "Water".

```
tagsWater = tagsDemo(contains(tagsDemo.Tags,"Water"),:)
```

```
tagsWater=48×1 table
                                                Tags
    ____________________________________________________________________________________________

    "OSIDemo_Flynn I.Water PH.42e7dc15-656a-5aa7-181c-6adee980d38a"
    "OSIDemo_Flynn I.Water Temperature.18d94851-7252-5c87-1fb3-d769fb949f15"
    "OSIDemo_Flynn II.Water PH.42e7dc68-656a-5aa7-181c-6adee980d38a"
    "OSIDemo_Flynn II.Water Temperature.18d9482c-7252-5c87-1fb3-d769fb949f15"
    "OSIDemo_GU1 Generator.Cooling Water Intake Temperature.2b0243be-4a97-5a48-1337-0e0b242c4795
    "OSIDemo_GU1 Generator.Cooling Water Output Temperature.7d1a79de-1957-5efc-1d55-aa5fcfccf36a
    "OSIDemo_GU1 Generator.Cooling Water Pressure.0c723cab-80c6-5630-13bf-38fdd7092768"
    "OSIDemo_GU1 Generator.Water in Oil.801b9776-f2fd-5a9b-2725-d2e66ad4284f"
    "OSIDemo_GU1 Turbine.Cooling Water Intake Temperature.f987f83c-7a17-5027-1492-0482f0ae04b9"
    "OSIDemo_GU1 Turbine.Cooling Water Output Temperature.62d8c804-2b82-53bc-09f1-660ffa00aa70"
    "OSIDemo_GU1 Turbine.Cooling Water Pressure Output.7c04ebc3-ae18-5207-0690-514252308c5b"
    "OSIDemo_GU1 Turbine.Water Flow.cf5b84c3-af11-5637-2317-f518e59b0c9f"
    "OSIDemo_GU1 Turbine.Water in Oil.562f5784-a814-5b11-3dd7-44b9735068ca"
    "OSIDemo_GU1 Turbine.Water pH Intake.2f31dbfe-6f9a-5005-083c-db8531de4d07"
    "OSIDemo_GU1 Turbine.Water pH Output.59993b66-79da-591e-1d17-ea5fad61a12b"
    "OSIDemo_GU2 Generator.Cooling Water Intake Temperature.2b0243d3-4a97-5a48-1337-0e0b242c4795
      ⋮
```

Identify Tag Based on Its Position in Existing Tag List

When narrowing your tag search, you can quickly single out a particular tag from a tag list by specifying its position in the list. This is often more convenient than identifying a tag by its name, which can sometimes be lengthy. For example, notice you can identify "OSIDemo_Flynn I.Water Temperature..." in the `tagsWater` list of the previous example by indicating the second entry in `tagsWater`.

```
tagFlynn1Water = tags(client, Name = tagsWater{2,:})
```

```
tagFlynn1Water=table
                                     Tags
    ________________________________________________________________________

    "OSIDemo_Flynn I.Water Temperature.18d94851-7252-5c87-1fb3-d769fb949f15"
```

Cleanup

When you are finished working with the PI Server, disconnect and remove the client by clearing its variable from the workspace.

```
clear client;
```

Read Data from a PI Server

This example shows you how to read data from an OSIsoft™ PI Server using the `read` method. The read capability of Industrial Communication Toolbox™ for PI provides a variety of options and flexible ways to get the data from your server. Running this example requires that an OSIsoft PI System is installed. The demo tags used in this example were provided by OSIsoft and may be downloaded from the following location:

https://learning.osisoft.com/asset-based-af-example-kits

The PI Server is capable of storing decades of real-time data from hundreds of assets. The MATLAB® interface for the PI System leverages the system's Asset Framework (AF) to access time series data of your assets.

Create Client/Server Connection and Retrieve Required Tags

Connect to a PI Server using the `piclient` function. In this example the Windows computer name is used as the PI AF Server name. Your situation may vary depending on PI System configuration.

```
host = getenv("COMPUTERNAME");
client = piclient(host);
```

Request a list of `tags` related to the asset of interest. For more detailed information see “Get Started Accessing Data from a PI Server” on page 21-74.

```
tagsTurbine = tags(client, Name = "OSIDemo_GU4 Turbine*")
```

```
tagsTurbine=17×1 table
                                              Tags
    ______________________________________________________________________________________

    "OSIDemo_GU4 Turbine.Bearing Temperature.c73204fa-e8f2-5513-1bc9-b5f097831d16"
    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-595f32095ce7"
    "OSIDemo_GU4 Turbine.Cooling Water Intake Temperature.f987e786-7a17-5027-1492-0482f0ae04b9"
    "OSIDemo_GU4 Turbine.Cooling Water Output Temperature.62d8d7be-2b82-53bc-09f1-660ffa00aa70"
    "OSIDemo_GU4 Turbine.Cooling Water Pressure Output.7c04f479-ae18-5207-0690-514252308c5b"
    "OSIDemo_GU4 Turbine.Hours Since Last Maintenance.ec8ae125-0ae5-5d45-1de6-0643ffac4983"
    "OSIDemo_GU4 Turbine.Lubricant Oil Intake Temperature.7fc3f299-b6ee-5e2e-3e7b-ca4ca59a9d9c"
    "OSIDemo_GU4 Turbine.Lubricant Oil Output Temperature.03c2ec47-7719-5f48-32b5-6c961d1a7912"
    "OSIDemo_GU4 Turbine.Lubricant Oil Pressure Output.9dd81865-26b3-57fb-0791-bf7a3cfea158"
    "OSIDemo_GU4 Turbine.Oil Level.201c4312-8852-50be-1d8b-297c216712ec"
    "OSIDemo_GU4 Turbine.Total Hours Running.4438ad34-9e54-5075-1c76-19e3ac3fb728"
    "OSIDemo_GU4 Turbine.Turbine Vibration.e4c9f243-d5c9-5f5c-3b82-ed8076592ff9"
    "OSIDemo_GU4 Turbine.Vane Angle.3ddc1860-a9cc-54ee-2d41-595eb92fc677"
    "OSIDemo_GU4 Turbine.Water Flow.cf5b9b79-af11-5637-2317-f518e59b0c9f"
    "OSIDemo_GU4 Turbine.Water in Oil.562f483e-a814-5b11-3dd7-44b9735068ca"
    "OSIDemo_GU4 Turbine.Water pH Intake.2f31c444-6f9a-5005-083c-db8531de4d07"
      ⋮
```

Read Latest Value of Tag

Read the most recent recorded value of a tag using the `read` method.

```
vibrationLatestTT = read(client, tagsTurbine.Tags(2))
```

```
vibrationLatestTT=1×3 timetable
              Time                                                    Tag
    _________________________    _______________________________________________________________

    21-December-2021 15:40:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
```

Read Values over Period of Time

To read values over a period of time, you first define a period. For example, to read values of a tag over the last two days, use the `DateRange` Name-Value pair to specify a starting `datetime` and ending `datetime`. Set the start date to two days ago.

```
startDate = datetime("now") - days(2);
```

Set the end date to now.

```
endDate = datetime("now");
```

Use these to specify the starting `datetime` and ending `datetime` in your request.

```
vibrationTwoDaysTT = read(client, tagsTurbine.Tags(2), DateRange = [startDate, endDate])
```

```
vibrationTwoDaysTT=559×3 timetable
              Time                                                    Tag
    _________________________    _______________________________________________________________

    19-December-2021 10:45:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 10:50:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 10:55:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:00:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:05:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:10:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:15:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:20:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:25:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:30:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:35:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:40:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:45:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:50:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 11:55:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
    19-December-2021 12:00:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
      ⋮
```

Read All Recorded Values of a Tag

To read all recorded values of a tag, it is useful to know when data recording began. You may use the `Earliest` Name-Value pair to determine this.

```
vibrationEarliestTT = read(client, tagsTurbine.Tags(2), Earliest = true)
```

```
vibrationEarliestTT=1×3 timetable
              Time                                                    Tag
    _________________________    _______________________________________________________________

    04-November-2021 20:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-
```

Notice the value of this tag at the earliest recorded time is a NaN. This is often the case for the first data point in a series as the PI Server indicates a status of Bad for this data point upon creation. You may take actions to exclude this from your data set if desired.

This earliest data point identifies the time of the first recorded value. You may now use this information to establish a starting `datetime` for your request.

```
startDate = datetime(vibrationEarliestTT.Time(1));
```

Set the ending `datetime` to now.

```
endDate = datetime("now", TimeZone = "local");
```

Depending on your system, this query may return a large amount of data. If you have an extensive history of data that makes this too slow or impractical, you may want to skip this step.

```
vibrationAllTT = read(client, tagsTurbine.Tags(2), DateRange = [startDate, endDate])
```

```
vibrationAllTT=13171×3 timetable
              Time                                             Tag
    ________________________    ____________________________________________________________

    04-November-2021 20:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 20:30:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 20:35:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 20:40:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 20:45:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 20:50:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 20:55:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:00:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:05:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:10:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:15:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:20:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:25:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:30:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:35:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:40:00    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
      ⋮
```

Reduce Dataset Using Linear Interpolation Provided by PI Server

Notice the large number of datapoints in the result of the previous step. You may reduce the dataset by using the `Interval` Name-Value pair. For example the following read requests data with an interval of 30 minutes. The `Interval` Name-Value pair requests the PI Server to perform linear interpolation on recorded values and provide results at the specified interval.

```
vibrationInterpolatedTT = read(client, tagsTurbine.Tags(2), DateRange = [startDate, endDate], In
```

```
vibrationInterpolatedTT=2247×3 timetable
              Time                                             Tag
    ________________________    ____________________________________________________________

    04-November-2021 20:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 20:55:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 21:55:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 22:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 22:55:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 23:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    04-November-2021 23:55:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    05-November-2021 00:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    05-November-2021 00:55:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    05-November-2021 01:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    05-November-2021 01:55:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    05-November-2021 02:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    05-November-2021 02:55:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    05-November-2021 03:25:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
    05-November-2021 03:55:43    "OSIDemo_GU4 Turbine.Bearing Vibration.64a3ce99-f31e-593c-3e29-5
      ⋮
```

Cleanup

When you are finished working with the PI Server, disconnect and remove the client by clearing its variable from the workspace.

```
clear client;
```

Process PI Data Using Common MATLAB Operations

This example shows you how to process PI data using common MATLAB timetable operations. Running this example requires that an OSIsoft™ PI System is installed. The demo tags used in this example were provided by OSIsoft and may be downloaded from the following location:

https://learning.osisoft.com/asset-based-af-example-kits

The PI Server is capable of storing decades of real-time data from hundreds of assets. The MATLAB® interface for the PI System leverages the system's Asset Framework (AF) to access time series data of your assets.

Create Client/Server Connection and Retrieve Required Tags

Connect to a PI Server using the `piclient` function. In this example the Windows computer name is used as the PI AF Server name. Your situation may vary depending on PI System configuration.

```
host = getenv("COMPUTERNAME");
client = piclient(host);
```

Request a list of `tags` related to the asset of interest. For more detailed information see “Get Started Accessing Data from a PI Server” on page 21-74.

```
tagsGenerator = tags(client, Name = "OSIDemo_GU1 Generator*")
```

```
tagsGenerator=28×1 table
                                                Tags
            ____________________________________________________________________________________

    "OSIDemo_GU1 Generator.Active Power.4db83f0a-ff87-5c67-385a-83cfe3ac560d"
    "OSIDemo_GU1 Generator.Axial Vibration.373a6144-442d-5e69-1079-5986bf866fa1"
    "OSIDemo_GU1 Generator.Bearing Temperature.3cfb845c-000c-5cf2-2ca0-b47dfdcfb4d7"
    "OSIDemo_GU1 Generator.Bearing Vibration.ac4f121c-0633-5a0d-3e26-e8d6a6249515"
    "OSIDemo_GU1 Generator.Cooling Water Intake Temperature.2b0243be-4a97-5a48-1337-0e0b242c4795"
    "OSIDemo_GU1 Generator.Cooling Water Output Temperature.7d1a79de-1957-5efc-1d55-aa5fcfccf36a"
    "OSIDemo_GU1 Generator.Cooling Water Pressure.0c723cab-80c6-5630-13bf-38fdd7092768"
    "OSIDemo_GU1 Generator.Core Temperature.79a1a7b5-5425-51ba-0bd6-ea1d6fbc4b86"
    "OSIDemo_GU1 Generator.Current Phase A.07815f38-c5b4-5abc-3d7c-2b0d4cd303ac"
    "OSIDemo_GU1 Generator.Current Phase B.7e11e986-d6e3-5f51-241a-23cc597bde31"
    "OSIDemo_GU1 Generator.Current Phase C.02c2ac50-14f7-5a5a-2e97-8b7d24851468"
    "OSIDemo_GU1 Generator.Frequency.8886141f-470a-514e-22d9-ef9e6aceaf95"
    "OSIDemo_GU1 Generator.Hours Since Last Maintenance.cb867a54-053b-5616-1a4a-dfa68f6b454c"
    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a312f1aa02e9"
    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2a1f0096c464"
    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-d2c1f98bf536"
      ⋮
```

Find All Tags from List Related to Voltage

Refine the list of tags using the `contains` function. This groups together all tags related to line voltages for use later in the example.

```
tagsVoltage = tagsGenerator(contains(tagsGenerator.Tags,"Voltage"),:)
```

```
tagsVoltage=3×1 table
                                        Tags
```

```
    ___________________________________________________________________________

    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a312f1aa02e9"
    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2a1f0096c464"
    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-d2c1f98bf536"
```

Read Latest Value of Multiple Tags

Read the latest value multiple tags using the `read` method and specifying a range of tags.

```
voltageLatestTT = read(client, tagsVoltage.Tags(1:3))
```

```
voltageLatestTT=3×3 timetable
            Time                                                       Tag
    _________________________    ____________________________________________________________

    21-December-2021 15:45:00    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-
    21-December-2021 15:45:00    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-
    21-December-2021 15:45:00    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-
```

Read All Recorded Values of Multiple Tags

To read all recorded values of a tag, it is useful to know when data recording began. You may use the `Earliest` Name-Value pair to determine this. Notice that all three of the tags from `tagsVoltage` are passed to the `read` method.

```
voltageEarliestTT = read(client, tagsVoltage.Tags(1:3), Earliest = true)
```

```
voltageEarliestTT=3×3 timetable
            Time                                                       Tag
    _________________________    ____________________________________________________________

    04-November-2021 20:25:12    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-
    04-November-2021 20:25:12    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-
    04-November-2021 20:25:12    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-
```

Notice the value of this tag at the earliest recorded time is a NaN. This is often the case for the first data point in a series as the PI Server indicates a status of Bad for this data point upon creation. You may take actions to exclude this from your data set if desired.

This earliest data point identifies the time of the first recorded value. You may now use this information to establish a starting `datetime` for your request..

```
startDate = datetime(voltageEarliestTT.Time(1));
endDate = datetime("now", TimeZone = "local");
```

Depending on your system, this query may return a large amount of data. If you have an extensive history of data that makes this too slow or impractical, you may want to skip this step.

```
voltageAllTT = read(client, tagsVoltage.Tags(1:3), DateRange = [startDate, endDate])
```

```
voltageAllTT=29815×3 timetable
            Time                                                       Tag
    _________________________    ____________________________________________________________
    04-November-2021 20:25:12    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    04-November-2021 20:25:12    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    04-November-2021 20:25:12    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-c
    04-November-2021 20:30:00    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    04-November-2021 20:30:00    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    04-November-2021 20:30:00    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-c
    04-November-2021 20:35:00    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    04-November-2021 20:40:00    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    04-November-2021 20:40:00    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    04-November-2021 20:45:00    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    04-November-2021 20:45:00    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    04-November-2021 20:50:00    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    04-November-2021 20:50:00    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    04-November-2021 20:50:00    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-c
    04-November-2021 20:55:00    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    04-November-2021 20:55:00    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-c
              ⋮
```

Reduce Dataset Using Linear Interpolation Provided by PI Server

Notice the large number of datapoints in the result of the previous step. You may reduce the dataset by using the `Interval` Name-Value pair. For example the following read requests data with an interval of 4 hours. The `Interval` Name-Value pair requests the PI Server to perform linear interpolation on recorded values and provide results at the specified interval.

```
voltageInterpolatedTT = read(client, tagsVoltage.Tags(1:3), DateRange = [startDate, endDate], In
```

```
voltageInterpolatedTT=843×3 timetable
              Time                                                   Tag
    _________________________    ____________________________________________________________

    04-November-2021 20:25:12    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    04-November-2021 20:25:12    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    04-November-2021 20:25:12    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-c
    05-November-2021 00:25:12    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    05-November-2021 00:25:12    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    05-November-2021 00:25:12    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-c
    05-November-2021 04:25:12    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    05-November-2021 04:25:12    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    05-November-2021 04:25:12    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-c
    05-November-2021 08:25:12    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    05-November-2021 08:25:12    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    05-November-2021 08:25:12    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-c
    05-November-2021 12:25:12    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    05-November-2021 12:25:12    "OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    05-November-2021 12:25:12    "OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-c
    05-November-2021 16:25:12    "OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
              ⋮
```

Unstack Values from One Timetable Variable to Multiple Variables

Unstack the timetable to distribute each line voltage as a timetable variable.

```
uVoltageTT = unstack(voltageInterpolatedTT,"Value","Tag",...
    "AggregationFunction",@(x)x(~isempty(x)),"VariableNamingRule","preserve")
```

```
uVoltageTT=281×3 timetable
              Time            OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    _________________________    _______________________________________________________________

    04-November-2021 20:25:12                                NaN
    05-November-2021 00:25:12                             3.2908
    05-November-2021 04:25:12                             3.2805
    05-November-2021 08:25:12                             3.3102
    05-November-2021 12:25:12                             3.3047
    05-November-2021 16:25:12                             3.2832
    05-November-2021 20:25:12                             3.2806
    06-November-2021 00:25:12                             3.3188
    06-November-2021 04:25:12                             3.2808
    06-November-2021 08:25:12                             3.2896
    06-November-2021 12:25:12                             3.3088
    06-November-2021 16:25:12                             3.3184
    06-November-2021 20:25:12                                3.3
    07-November-2021 00:25:12                             3.3008
    07-November-2021 04:25:12                             3.3188
    07-November-2021 08:25:12                             3.3104
      ⋮
```

Fill Missing Values

Notice the timetable in the previous step contains some NaN values. This happens when datapoints of a timetable are not all sampled at the same interval. Use the `fillmissing` function to correct this using linear interpolation.

```
[cVoltageTT,~] = fillmissing(uVoltageTT,"linear")
```

```
cVoltageTT=281×3 timetable
              Time            OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    _________________________    _______________________________________________________________

    04-November-2021 20:25:12                             3.3011
    05-November-2021 00:25:12                             3.2908
    05-November-2021 04:25:12                             3.2805
    05-November-2021 08:25:12                             3.3102
    05-November-2021 12:25:12                             3.3047
    05-November-2021 16:25:12                             3.2832
    05-November-2021 20:25:12                             3.2806
    06-November-2021 00:25:12                             3.3188
    06-November-2021 04:25:12                             3.2808
    06-November-2021 08:25:12                             3.2896
    06-November-2021 12:25:12                             3.3088
    06-November-2021 16:25:12                             3.3184
    06-November-2021 20:25:12                                3.3
    07-November-2021 00:25:12                             3.3008
    07-November-2021 04:25:12                             3.3188
    07-November-2021 08:25:12                             3.3104
      ⋮
```

View Each Voltage in a Separate Timetable

View voltage AB in its own timetable if you like.

```
lineVoltageAB = cVoltageTT.Properties.VariableNames{1};
vabTT = timetable(cVoltageTT.(lineVoltageAB)(:), 'RowTimes', cVoltageTT.Time(:), 'VariableNames'
```

```
vabTT=281×1 timetable
               Time              OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a
    _________________________    _______________________________________________________________

    04-November-2021 20:25:12                                   3.3011
    05-November-2021 00:25:12                                   3.2908
    05-November-2021 04:25:12                                   3.2805
    05-November-2021 08:25:12                                   3.3102
    05-November-2021 12:25:12                                   3.3047
    05-November-2021 16:25:12                                   3.2832
    05-November-2021 20:25:12                                   3.2806
    06-November-2021 00:25:12                                   3.3188
    06-November-2021 04:25:12                                   3.2808
    06-November-2021 08:25:12                                   3.2896
    06-November-2021 12:25:12                                   3.3088
    06-November-2021 16:25:12                                   3.3184
    06-November-2021 20:25:12                                      3.3
    07-November-2021 00:25:12                                   3.3008
    07-November-2021 04:25:12                                   3.3188
    07-November-2021 08:25:12                                   3.3104
      ⋮
```

View voltage AC in its own timetable.

```
lineVoltageAC = cVoltageTT.Properties.VariableNames{2};
vacTT = timetable(cVoltageTT.(lineVoltageAC)(:), 'RowTimes', cVoltageTT.Time(:), 'VariableNames'
```

```
vacTT=281×1 timetable
               Time              OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2
    _________________________    _______________________________________________________________

    04-November-2021 20:25:12                                   3.3298
    05-November-2021 00:25:12                                   3.3053
    05-November-2021 04:25:12                                   3.2808
    05-November-2021 08:25:12                                   3.3194
    05-November-2021 12:25:12                                   3.3092
    05-November-2021 16:25:12                                   3.3094
    05-November-2021 20:25:12                                   3.3188
    06-November-2021 00:25:12                                     3.32
    06-November-2021 04:25:12                                   3.2812
    06-November-2021 08:25:12                                   3.2906
    06-November-2021 12:25:12                                   3.3192
    06-November-2021 16:25:12                                   3.2806
    06-November-2021 20:25:12                                   3.3188
    07-November-2021 00:25:12                                   3.2992
    07-November-2021 04:25:12                                   3.2908
    07-November-2021 08:25:12                                   3.3196
      ⋮
```

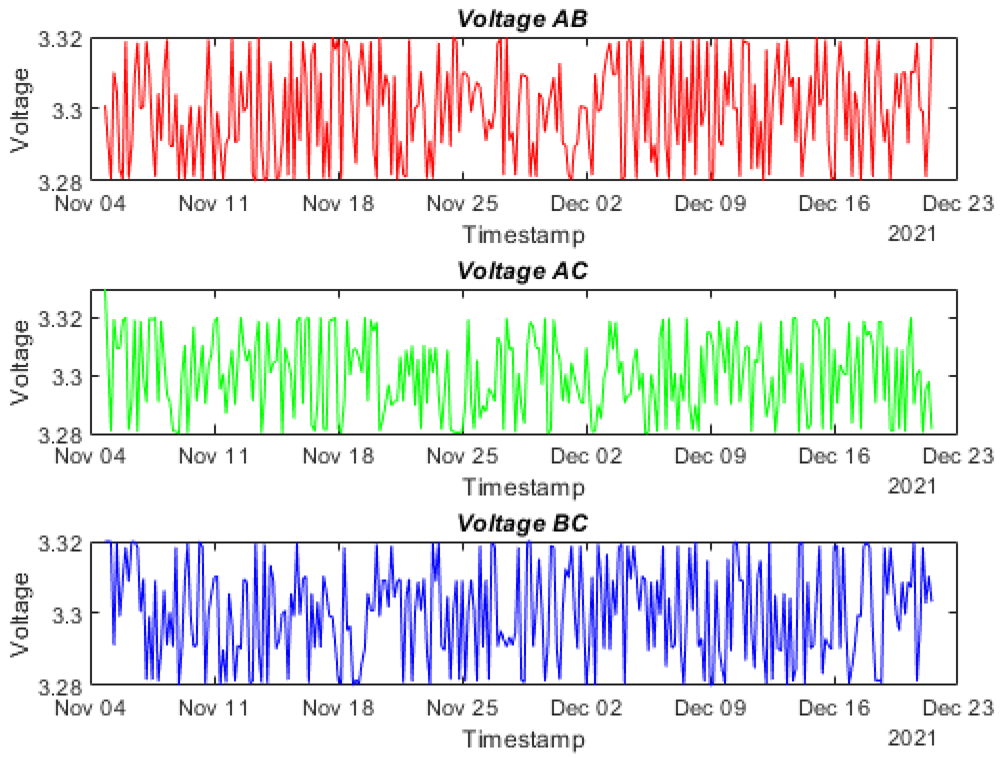
View voltage BC in its own timetable.

```
lineVoltageBC = cVoltageTT.Properties.VariableNames{3};
vbcTT = timetable(cVoltageTT.(lineVoltageBC)(:), 'RowTimes', cVoltageTT.Time(:), 'VariableNames'
vbcTT=281×1 timetable
              Time              OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-d
    _________________________    _______________________________________________________________

    04-November-2021 20:25:12                                  3.32
    05-November-2021 00:25:12                                  3.32
    05-November-2021 04:25:12                                  3.32
    05-November-2021 08:25:12                                3.2912
    05-November-2021 12:25:12                                3.3196
    05-November-2021 16:25:12                                3.2992
    05-November-2021 20:25:12                                3.3096
    06-November-2021 00:25:12                                3.3184
    06-November-2021 04:25:12                                3.3088
    06-November-2021 08:25:12                                3.3198
    06-November-2021 12:25:12                                  3.32
    06-November-2021 16:25:12                                3.3184
    06-November-2021 20:25:12                                3.3004
    07-November-2021 00:25:12                                3.3097
    07-November-2021 04:25:12                                3.2816
    07-November-2021 08:25:12                                3.2992
       ⋮
```

Visualize Tag Values of Voltages

To visualize values of interest, voltages from the timetables can be plotted over time for further analysis.

```
subplot(3, 1, 1)
plot(vabTT.Time, vabTT.("OSIDemo_GU1 Generator.Line Voltage AB.373eb947-c651-5aef-1948-a"), "r")
title("{\itVoltage AB}", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Voltage")
subplot(3, 1, 2)
plot(vacTT.Time, vacTT.("OSIDemo_GU1 Generator.Line Voltage AC.a809d1f1-c08f-54f9-0915-2"), "g")
title("{\itVoltage AC}", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Voltage")
subplot(3, 1, 3)
plot(vbcTT.Time, vbcTT.("OSIDemo_GU1 Generator.Line Voltage BC.90cc345e-520f-5284-187b-d"), "b")
title("{\itVoltage BC}", "FontWeight", "bold")
xlabel("Timestamp")
ylabel("Voltage")
```



Cleanup

When you are finished working with the PI Server, disconnect and remove the client by clearing its variable from the workspace.

```
clear client;
```

Get Started with MQTT

This example shows how to establish a secure connection in MATLAB with an MQTT broker and communicate with the MQTT broker.

ThingSpeak™ is used as the broker in this example.

Message Queuing Telemetry Transport (MQTT) is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth.

ThingSpeak is an IoT analytics platform service that allows you to aggregate, visualize, and analyze live data streams in the cloud. You can send data to ThingSpeak from your devices, create instant visualization of live data, and send alerts.

Set Up the Broker and Get a Root Certificate

To establish a connection with ThingSpeak, see “Create a ThingSpeak MQTT Device” (ThingSpeak). After creating the ThingSpeak MQTT device, you can get its Client ID, Username and Password from it. Assign those values in MATLAB®.

```
clientID = "Your Client ID";
userName = "Your Username";
password = "Your Password";
```

Download the root certificate from thingspeak.com as described in How to Download Root Certificate for Use With Industrial Communication Toolbox MQTT Functions. Get the path of the downloaded root certificate. The location and file name extension depends on the browser you use. For example, using Edge you might set `rootCert` like this:

```
rootCert = "C:\Downloads\DigiCert Global Root CA.crt";
```

The certificate saved from Firefox might have the file extension `.pem`.

Create an MQTT Client and Connect to the Broker with SSL

Prepare the broker address and port number you want to connect. In this case, set up a secure connection to ThingSpeak via SSL with an appropriate port number.

```
brokerAddress = "ssl://mqtt3.thingspeak.com";
port = 8883;
```

Create an MQTT client using the `mqttclient` function.

```
mqClient = mqttclient(brokerAddress, Port = port, ClientID = clientID,...
           Username = userName, Password = password, CARootCertificate = rootCert);
```

Note that the `Connected` property indicates the connection to the broker has been established.

```
mqClient.Connected
```

```
ans = int32
    1
```

Subscribe to a Topic

With the connected MQTT client, use the `subscribe` function to subscribe to the topic of interest. The displayed table shows the subscribed topic. For details about the topics to subscribe to in ThingSpeak, see Subscribe to a Channel Field Feed (ThingSpeak).

```
topicToSub = "channels/1393455/subscribe/fields/field2";
subscribe(mqClient, topicToSub)
```

```
ans=1×3 table
                  Topic                     QualityOfService    Callback
    __________________________________________    ________________    ________

    "channels/1393455/subscribe/fields/field2"           0              ""
```

Write to a Topic

To verify that the subscription is successful, make sure a message written to the subscribed topic is received by the MQTT client.

Use the `write` function to write messages to the topic of interest. For details about the topics to write to in ThingSpeak, see Publish to a Channel Field Feed (ThingSpeak).

```
topicToWrite = "channels/1393455/publish/fields/field2";
msg = "70";
write(mqClient, topicToWrite, msg)
```

Peek at the MQTT Client

Use the `peek` function to view the most recently received message for all subscribed topics in the MQTT client. The displayed timetable indicates the MQTT client has successfully received the message from the broker.

```
peek(mqClient)
```

```
ans=1×2 timetable
            Time                              Topic                         Data
    ____________________    __________________________________________    ____

    06-Jan-2022 10:42:29    "channels/1393455/subscribe/fields/field2"    "70"
```

Close the MQTT Client

Close the connection to ThingSpeak by clearing the MQTT client variable from the workspace.

```
clear mqClient
```

Get Data from Subscribed Topics in an MQTT Client

This example shows how to get data from subscribed topics in an MQTT client.

ThingSpeak™ is used as the broker in this example.

Message Queuing Telemetry Transport (MQTT) is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth.

ThingSpeak is an IoT analytics platform service that allows you to aggregate, visualize, and analyze live data streams in the cloud. You can send data to ThingSpeak from your devices, create instant visualization of live data, and send alerts.

Create an MQTT Client and Connect to the Broker

Set up a ThingSpeak broker and get Client ID, Username, and Password from it. Assign those values in MATLAB®.

```
clientID = "Your Client ID";
userName = "Your Username";
password = "Your Password";
```

Download the root certificate from thingspeak.com as described in How to Download Root Certificate for Use With Industrial Communication Toolbox MQTT Functions. Get the path of the downloaded root certificate. The location and file name extension depends on the browser you use. For example, using Edge you might set `rootCert` like this:

```
rootCert = "C:\Downloads\DigiCert Global Root CA.crt";
```

The certificate saved from Firefox might have the file extension `.pem`.

Establish a secure connection to ThingSpeak with an appropriate port number using the `mqttclient` function.

```
brokerAddress = "ssl://mqtt3.thingspeak.com";
port = 8883;
mqClient = mqttclient(brokerAddress, Port = port, ClientID = clientID,...
           Username = userName, Password = password, CARootCertificate = rootCert);
```

Subscribe to a Topic

Use the `subscribe` function to subscribe to the topic of interest. After subscription, the MQTT client in MATLAB receives and stores all the data written to the topic of interest.

```
topicToSub = "channels/1393455/subscribe/fields/field2";
subscribe(mqClient, topicToSub)
```

```
ans=1×3 table
                     Topic                        QualityOfService    Callback
    __________________________________________    ________________    ________

    "channels/1393455/subscribe/fields/field2"           0               ""
```

Write to the Subscribed Topic

Use the `write` function to write messages to the topic of interest. In this case, 3 messages are written to the subscribed topic. Pause for a few seconds after each `write` to avoid violating the rate limits in ThingSpeak.

```
topicToWrite = "channels/1393455/publish/fields/field2";
msg1 = "70";
msg2 = "73";
msg3 = "69";
write(mqClient, topicToWrite, msg1)
pause(2)
write(mqClient, topicToWrite, msg2)
pause(2)
write(mqClient, topicToWrite, msg3)
pause(2)
```

Read Received Data from the Subscribed Topic

Use the `read` function to read all data received from the subscribed topic into a timetable. Note that `read` removes all the data stored in the subscribed topic you just read from.

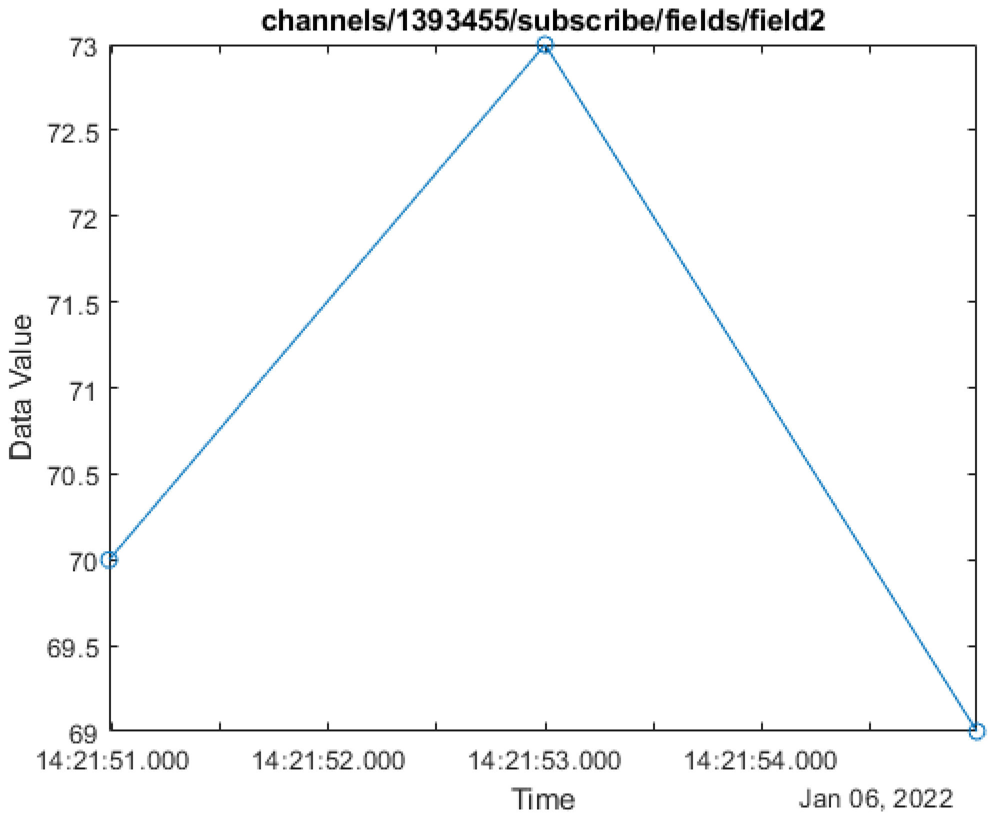
```
dataTT = read(mqClient)
```

```
dataTT=3×2 timetable
            Time                            Topic                           Data
    ____________________    __________________________________________    ____

    06-Jan-2022 14:21:50    "channels/1393455/subscribe/fields/field2"    "70"
    06-Jan-2022 14:21:52    "channels/1393455/subscribe/fields/field2"    "73"
    06-Jan-2022 14:21:54    "channels/1393455/subscribe/fields/field2"    "69"
```

Visualize the Received Data

To visualize the information, plot the received data from the subscribed topic.

```
t = dataTT.Time;
data = str2double(dataTT.Data);
plot(t, data, 'o-')
title(topicToSub)
xlabel("Time")
ylabel("Data Value")
```



Close the MQTT Client

Close access to ThingSpeak by clearing the MQTT client variable from the workspace.

```
clear mqClient
```

Subscribe to an MQTT Topic with a Callback Function

This example shows how to use an MQTT client to subscribe to a topic with a callback function.

ThingSpeak™ is used as the broker in this example.

Message Queuing Telemetry Transport (MQTT) is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth.

ThingSpeak is an IoT analytics platform service that allows you to aggregate, visualize, and analyze live data streams in the cloud. You can send data to ThingSpeak from your devices, create instant visualization of live data, and send alerts.

Create an MQTT Client and Connect to the Broker

Set up a ThingSpeak broker and get Client ID, Username, and Password from it. Assign those values in MATLAB®.

```
clientID = "Your Client ID";
userName = "Your Username";
password = "Your Password";
```

Download the root certificate from thingspeak.com as described in How to Download Root Certificate for Use With Industrial Communication Toolbox MQTT Functions. Get the path of the downloaded root certificate. The location and file name extension depends on the browser you use. For example, using Edge you might set `rootCert` like this:

```
rootCert = "C:\Downloads\DigiCert Global Root CA.crt";
```

The certificate saved from Firefox might have the file extension `.pem`.

Establish a secure connection to ThingSpeak with an appropriate port number using the `mqttclient` function.

```
brokerAddress = "ssl://mqtt3.thingspeak.com";
port = 8883;
mqClient = mqttclient(brokerAddress, Port = port, ClientID = clientID,...
           Username = userName, Password = password, CARootCertificate = rootCert);
```

Subscribe to a Topic with a Callback Function

To subscribe with a callback function, create a callback function named `showmessage`. The `showmessage` function prints the received data and corresponding topic when triggered.

Use the `subscribe` function to subscribe to the topic of interest. Use a Name-Value pair argument to assign the callback function at the same time. The displayed table shows the subscribed topic and the corresponding callback function.

```
topicToSub = "channels/1393455/subscribe/fields/field2";
subscribe(mqClient, topicToSub, Callback = "showmessage")
```

```
ans=1×3 table
                      Topic                            QualityOfService       Callback
```

```
    __________________________________________    ________________    _____________

    "channels/1393455/subscribe/fields/field2"           0            "showmessage"
```

Write to the Subscribed Topic

To trigger the callback function, the MQTT client needs to receive messages for the subscribed topic. Use the `write` function to write messages to the subscribed topic.

```
topicToWrite = "channels/1393455/publish/fields/field2";
msg = "70";
write(mqClient, topicToWrite, msg)
```

Trigger Callback Function

Pause to allow the message to transfer from the MQTT client, to the MQTT broker, and back to the client.

```
pause(2)
```

When the MQTT client receives the message from the subscribed topic, the callback function `showmessage` is automatically triggered. The following context is printed in the MATLAB Command Window.

```
Topic: channels/1393455/subscribe/fields/field2, Message: 70
```

Close the MQTT Client

Close access to ThingSpeak by clearing the MQTT client variable from the workspace.

```
clear mqClient
```

Subscribe to an MQTT Wildcard Topic

This example shows how to use an MQTT client to subscribe to a wildcard topic.

ThingSpeak™ is used as the broker in this example.

Message Queuing Telemetry Transport (MQTT) is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth.

ThingSpeak is an IoT analytics platform service that allows you to aggregate, visualize, and analyze live data streams in the cloud. You can send data to ThingSpeak from your devices, create instant visualization of live data, and send alerts.

Create an MQTT Client and Connect to the Broker

Set up a ThingSpeak broker and get Client ID, Username, and Password from it. Assign those values in MATLAB®.

```
clientID = "Your Client ID";
userName = "Your Username";
password = "Your Password";
```

Download the root certificate from thingspeak.com as described in How to Download Root Certificate for Use With Industrial Communication Toolbox MQTT Functions. Get the path of the downloaded root certificate. The location and file name extension depend on the browser you use. For example, using Edge you might set `rootCert` like this:

```
rootCert = "C:\Downloads\DigiCert Global Root CA.crt";
```

The certificate saved from Firefox might have the file extension `.pem`.

Establish a secure connection to ThingSpeak with an appropriate port number using the `mqttclient` function.

```
brokerAddress = "ssl://mqtt3.thingspeak.com";
port = 8883;
mqClient = mqttclient(brokerAddress, Port = port, ClientID = clientID,...
           Username = userName, Password = password, CARootCertificate = rootCert);
```

Subscribe to a Wildcard Topic

To subscribe to all the topics under a certain hierarchy, use the `subscribe` function with a wildcard to make the subscription easier. The displayed table shows the wildcard topic has been subscribed successfully.

```
topicWildcard = "channels/1393455/subscribe/fields/+";
subscribe(mqClient, topicWildcard)
```

```
ans=1×3 table
                    Topic                      QualityOfService    Callback
    _______________________________________    ________________    ________

    "channels/1393455/subscribe/fields/+"             0               ""
```

Write to Different Topics Under the Wildcard

To verify that the wildcard subscription is successful, make sure the messages written to different topics under the wildcard subscription are received by the MQTT client.

Use the `write` function to write messages to different topics under the wildcard. Pause for a few seconds after each `write` to avoid violating the rate limits in ThingSpeak.

```
topicToWrite1 = "channels/1393455/publish/fields/field1";
topicToWrite2 = "channels/1393455/publish/fields/field2";
msg1 = "60";
msg2 = "30";
write(mqClient, topicToWrite1, msg1)
pause(2)
write(mqClient, topicToWrite2, msg2)
pause(2)
```

Peek at the MQTT Client

Use the `peek` function to view the most recently received data for all subscribed topics. The displayed timetable indicates that the messages under the wildcard have been received successfully.

```
peek(mqClient)
```

```
ans=2×2 timetable
            Time                              Topic                            Data
    ____________________    __________________________________________    ____

    06-Jan-2022 13:36:40    "channels/1393455/subscribe/fields/field1"    "60"
    06-Jan-2022 13:36:43    "channels/1393455/subscribe/fields/field2"    "30"
```

Close the MQTT Client

Close access to ThingSpeak by clearing the MQTT client variable from the workspace.

```
clear mqClient
```