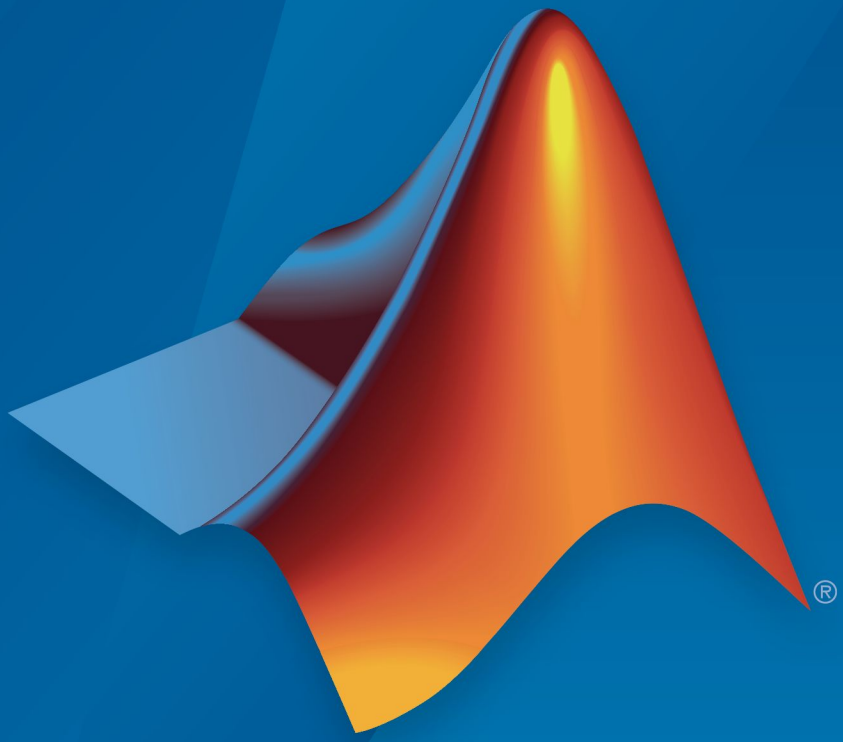
OPC Toolbox™

User s Guide



MATLAB® & SIMULINK®

R2016a

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.
3 Apple Hill Drive
Natick, MA 01760-2098

OPC Toolbox™ User's Guide

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)

Contents

Getting Started

1 Introduction

2 Quick Start: Using OPC Data Access Functions

3 Quick Start: Using the OPC Data Access Explorer

4 Quick Start: Using OPC Historical Data Access Functions

Data Access User's Guide

5 Introduction to OPC Data Access (DA)

6 Using OPC Toolbox Data Access Objects

7 Reading, Writing, and Logging OPC Data

8 Working with OPC Data

9 Using Events and Callbacks

10 Using the OPC Toolbox Block Library

Properties — Alphabetical List

11

Historical Data Access User's Guide

Introduction to OPC Historical Data Access (HDA)

12

13 Using OPC Toolbox HDA Client Objects

14 Reading OPC Historical Data

15 Working with OPC HDA Data Objects

16 OPC HDA and UA Classes — Alphabetical List

Unified Architecture User's Guide

17 OPC Unified Architecture (UA)

OPC Information Reference

A OPC Quality Strings

Getting Started

1

Introduction

OPC Toolbox Product Description

Read and write data from OPC servers and data historians

OPC Toolbox provides access to live and historical OPC data directly from MATLAB® and Simulink®. You can read, write, and log OPC data from devices, such as distributed control systems, supervisory control and data acquisition systems, and programmable logic controllers. OPC Toolbox lets you work with data from live servers and data historians that conform to the OPC Data Access (DA) standard, the OPC Historical Data Access (HDA) standard, and the OPC Unified Architecture (UA) standard.

The product includes Simulink blocks that let you model online supervisory control and perform hardware-in-the-loop controller testing.

Key Features

- OPC Unified Architecture (UA) standard v1.02 support
- OPC Foundation Data Access (DA) standard v2.05a support
- OPC Foundation Historical Data Access (HDA) standard v1.20 support
- Simultaneous data logging and numerical processing
- Simultaneous connections to multiple OPC servers
- Access to historical data for analysis and statistical processing
- Communication with OPC servers using synchronous or asynchronous operations

Overview of OPC, Servers, and the Toolbox

In this section...

About OPC Toolbox Software

OPC 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 12-7 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 Toolbox Data Access Objects" on page 6-2 describes how to create and configure `dagroup` objects and add `daitem` objects to a group.

Using OPC Toolbox 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 7-15 describes how to log data using the OPC Toolbox 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 Toolbox 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 Toolbox block library to use live OPC data as inputs to your model and update the OPC server with your model outputs. The OPC Toolbox 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 OPC Toolbox block library.

About OPC

Open Process Control (OPC), also known as OLE for Process Control, is a series of seven specifications defined by the OPC Foundation (http://www.opcfoundation.org) for supporting open connectivity in industrial automation. 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 OPC 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

OPC Toolbox software is 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.

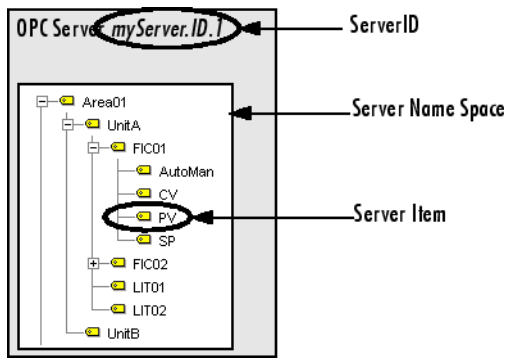


Figure 1-1.

A *server item* represents a value on the OPC server that a client may 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 Figure 1-1, , 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 12-8.

System Requirements

OPC Toolbox software provides the 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 for OPC Toolbox Software” on page 1-9.

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

```
help opchdaserverinfo
```

OPC 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 OPC 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 Toolbox object properties.

```
opchelp EventLog
```

Set Up for OPC Toolbox Software

In this section...

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 OPC 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 OPC Toolbox 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

The OPC Foundation has provided 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 OPC Toolbox software.

To install the OPC Foundation Core Components, you 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 Toolbox 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, OPC 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 OPC 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 OPC 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 1-11 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 1-17 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

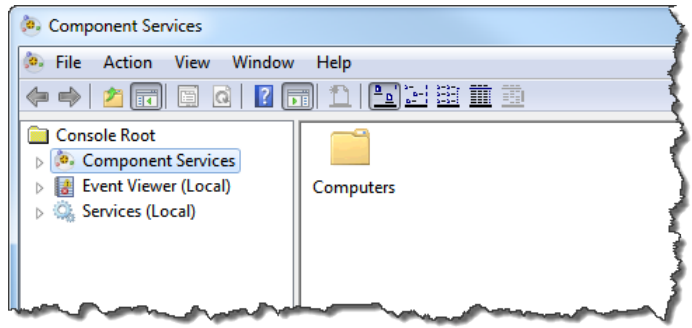
To configure DCOM to use named user security, you will have to ensure that both the server machine and client machine have a common user who is granted DCOM access rights on both the server and client machines. You should consult the following sections for information on configuring each machine:

- "OPC Server Machine Configuration" on page 1-11 provides the steps that you must perform on each of the machines providing OPC servers.
- "Client Machine Configuration" on page 1-14 provides the steps that you must perform on the machine that will run MATLAB and OPC Toolbox software.

OPC Server Machine Configuration

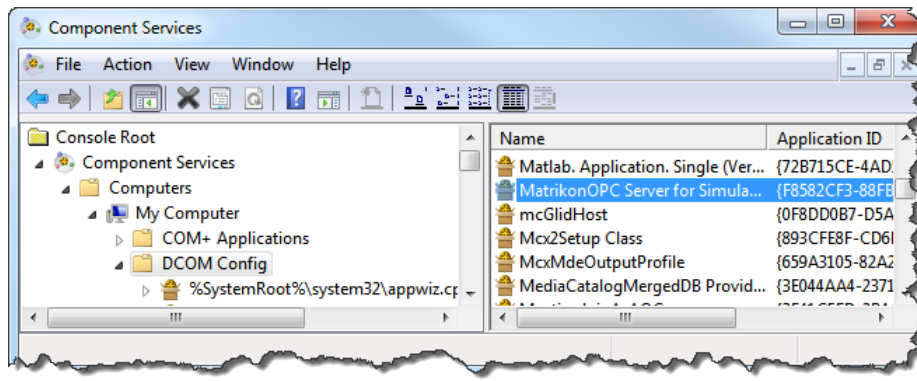
On the machines hosting the OPC servers, perform the following steps:

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`.

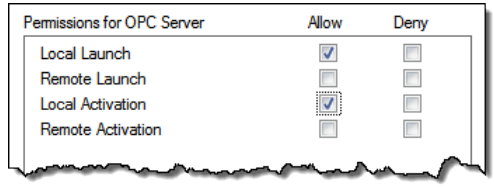
4 Locate your OPC server in the `DCOM Config` list. The example below shows the Matrikon™ OPC Server for Simulation.



5 Right-click the OPC server object, and choose **Properties**.

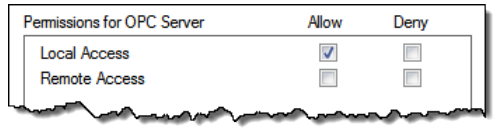
6 In the **General** tab, ensure that the Authentication Level is set to `Default` or to `Connect`.

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.



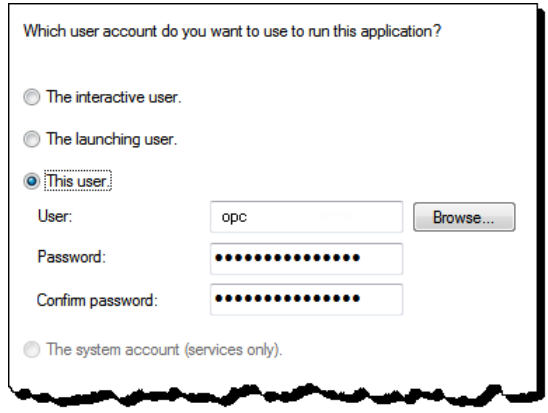
Click **OK** to dismiss the Local Launch and Activation Permissions dialog box.

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



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).



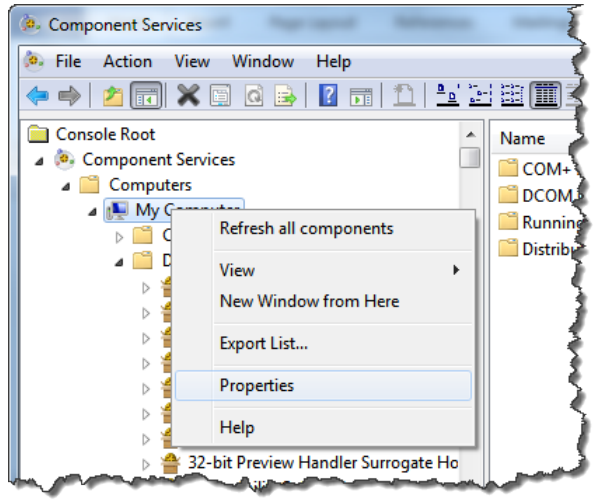
10 If the OPC server runs as a service, make sure that the service runs as the `opc` user (created in step 1) and not as the system account. Consult your system administrator for information on how to configure a service to run as a specific user.

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

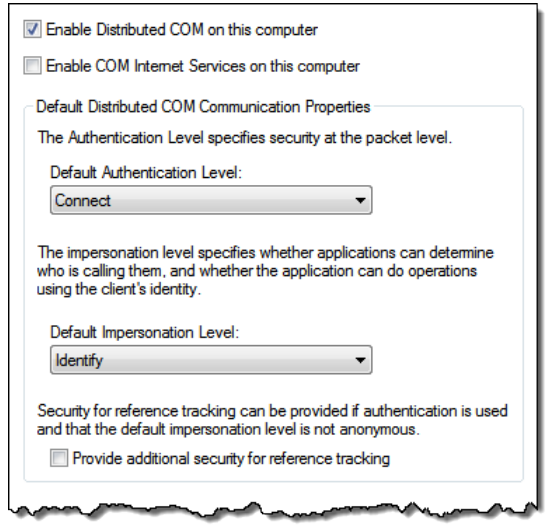
Client Machine Configuration

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

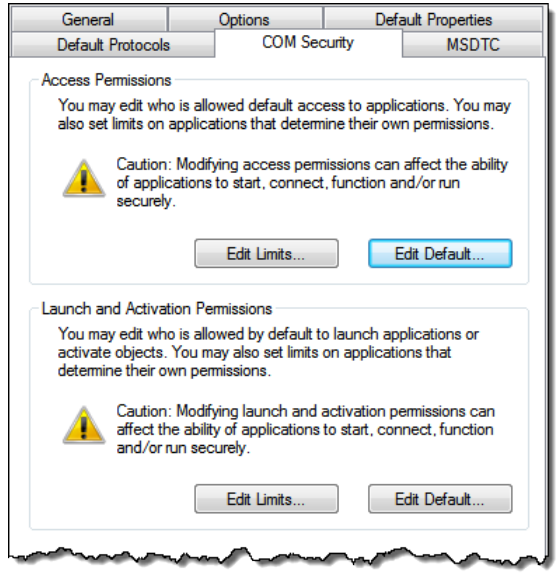
1. On the client machine(s), create the identical local user with the same name and password permissions as you set up in step 1 of "OPC Server Machine Configuration" on page 1-11.
2. Select **Start > Control Panel**. Double-click `Administrative Tools` and then double-click `Component Services`. The Component Services dialog appears.
3. Browse to `Component Services > Computers > My Computer`. Right-click `My Computer` and select **Properties**.



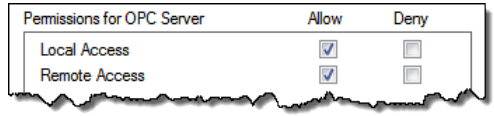
4. Click the **Default Properties** tab, and ensure that:
   - Enable Distributed COM is checked
   - Default Authentication Level is set to `Connect`
   - Default Impersonation Level is set to `Identify`



5 Click the **COM Security** tab.



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.

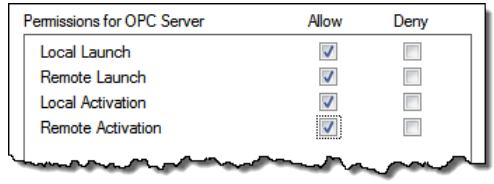


Click **OK** to close the Default Access Permissions dialog box.

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).



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.

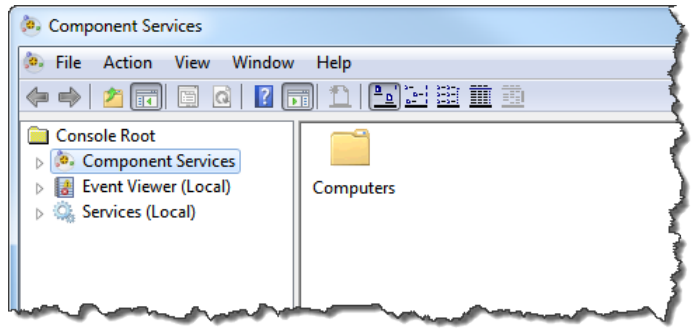
Your local client machine and server applications are now configured to use the same username when the server attempts to establish a connection back to the client.

Configure DCOM to Use No Security

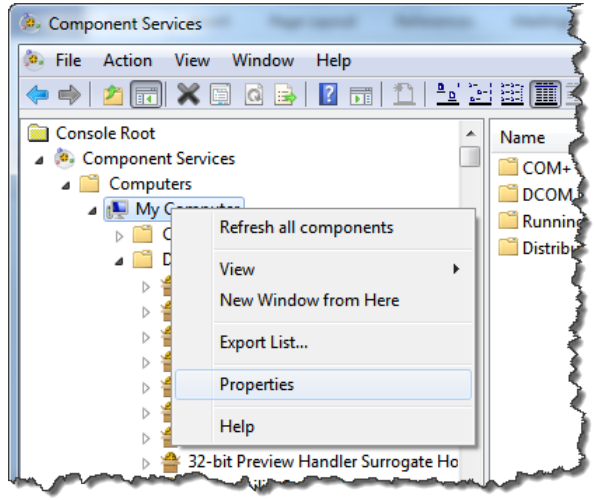
Caution You should not use this option if you are not in a completely trusted network. Turning off DCOM security means that any user on the network can launch any COM object on your local machine. Consult your network administrator before following these instructions.

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

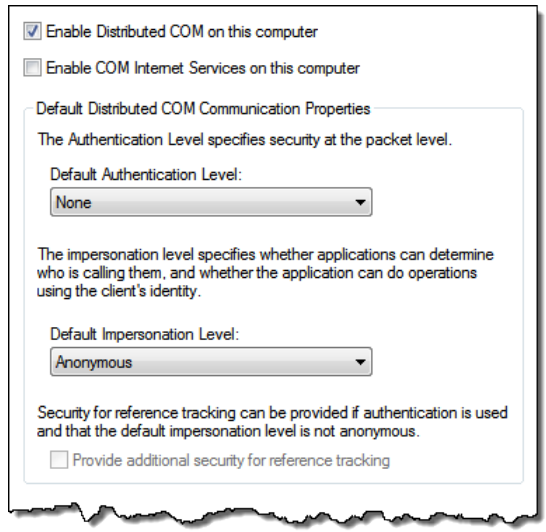
1. Ensure that the `Guest` user account is enabled. (The `Guest` account is disabled by default on Windows 7 machines). Consult your system administrator for information on how to enable the `Guest` account.
2. Select **Start > Control Panel**. Double-click `Administrative Tools` and then double-click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



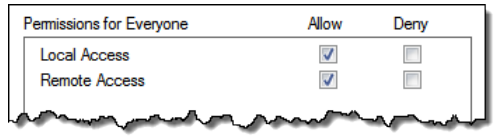
3. Browse to `Component Services > Computers > My Computer`. Right-click `My Computer` and select **Properties**.



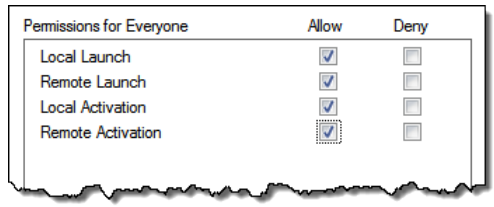
4 In the **Default Properties** tab, make sure that Enable Distributed COM On This Computer is selected. Select None as the Default Authentication Level, and Anonymous as the Default Impersonation Level.



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.



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).



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 the Matrikon OPC Simulation Server

All examples in this guide and in the OPC Toolbox online help make use of a Matrikon simulation server that you can download free of charge from:

```
http://www.matrikonopc.com
```

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

To install the Matrikon OPC Simulation Server, follow the installation instructions with the software. When prompted for a server ID, use the standard server ID assigned to the Simulation Server (`Matrikon.OPC.Simulation`).

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, OPC 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, OPC 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 OPC 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 OPC Foundation Sample Servers

OPC Toolbox includes various examples showing you how to communicate with OPC UA servers. To run those examples you need to install the OPC Foundation Sample Servers. For more information, see Install an OPC UA Simulation Server for OPC UA Examples.

Troubleshooting

In this section...

Troubleshooting Introduction

If you are unable to establish a connection to an OPC server, the following sections might help you to identify 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 1-10.

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 1-9.

“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 1-9.

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 1-10.

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 1-10.

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 1-10.

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 OPC Toolbox software client on the local machine. Review the section on “Configure DCOM” on page 1-10.

Quick Start: Using OPC Data Access Functions

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

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

Access Data at Command Line

In this section...

DA Programming Overview

This section illustrates the basic steps to create an OPC Toolbox 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 the Matrikon OPC Simulation Server" on page 1-19. 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 2-4.

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 Toolbox 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 string, 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 Toolbox 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 Toolbox 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 Toolbox 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 6-5 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

2-6. 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 string to limit the returned results. The string 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 string. 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 6-7.

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 7-11.

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 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. OPC Toolbox software allows you to log all 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 Toolbox 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 Toolbox objects at the MATLAB command line or from your own functions, you must remove them from the OPC 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 Toolbox object management is discussed in detail in “Delete Objects” on page 6-24.

Quick Start: Using the OPC Data Access Explorer

The best way to learn about the capabilities of OPC Toolbox software 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 OPC Data Access Explorer

In this section...

Precedure Overview

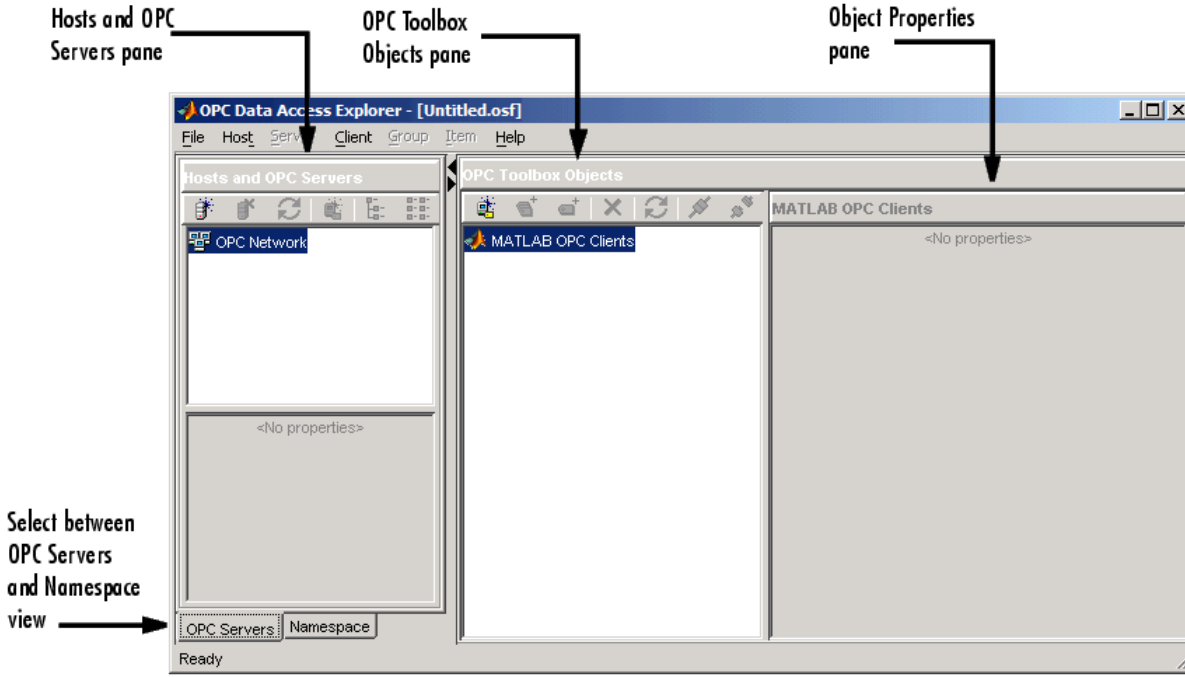
This section illustrates the basic steps required to create an OPC Toolbox Data Access application by visualizing the Triangle Wave and Saw-toothed Wave signals provided with the Matrikon OPC Simulation Server. The application logs data to memory and plots that data, highlighting uncertain or bad data 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 the Matrikon OPC Simulation Server" on page 1-19. 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 Toolbox 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 3-6.

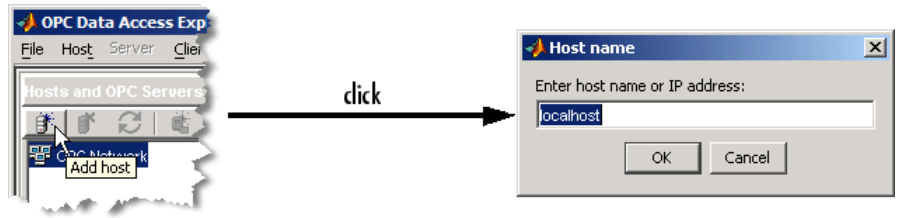
The first piece of information that you require is the hostname of the server computer. The hostname (a descriptive name like `PlantServer` or an IP address such as `192.168.16.32`) qualifies that computer on the network, and is used by the OPC Data Access protocols to determine the available OPC servers on that computer, and to communicate with the computer to establish a connection to the server. In any OPC Toolbox 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 text string, 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:

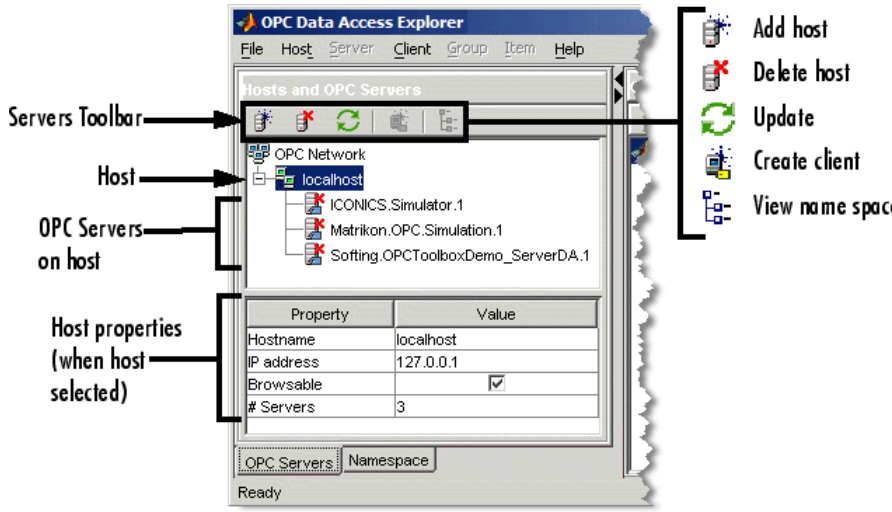
1. In the **Hosts and OPC Servers** pane, click the **Add host** icon to open the Host name dialog, shown below.



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.



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
```

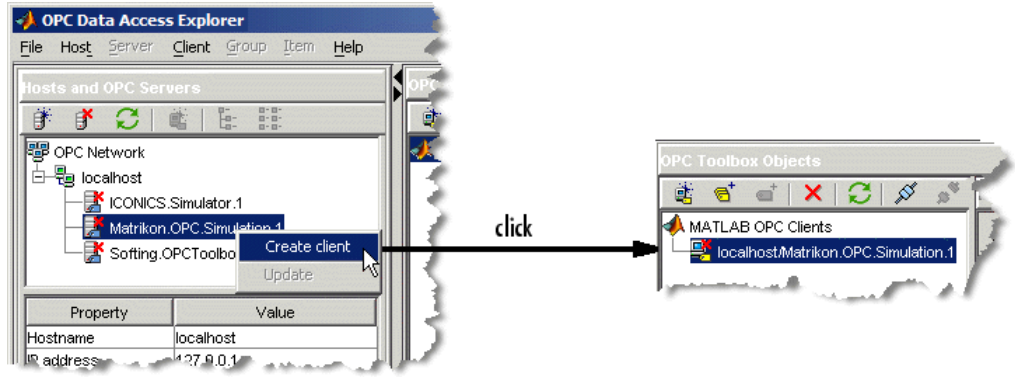
```
allServers =
     Matrikon.OPC.Simulation.1
     ICONICS.Simulator.1
     Softing.OPCToolboxDemo_ServerDA.1
```

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 Toolbox object hierarchy, see "Toolbox Object Hierarchy for the Data Access Standard" on page 6-2.

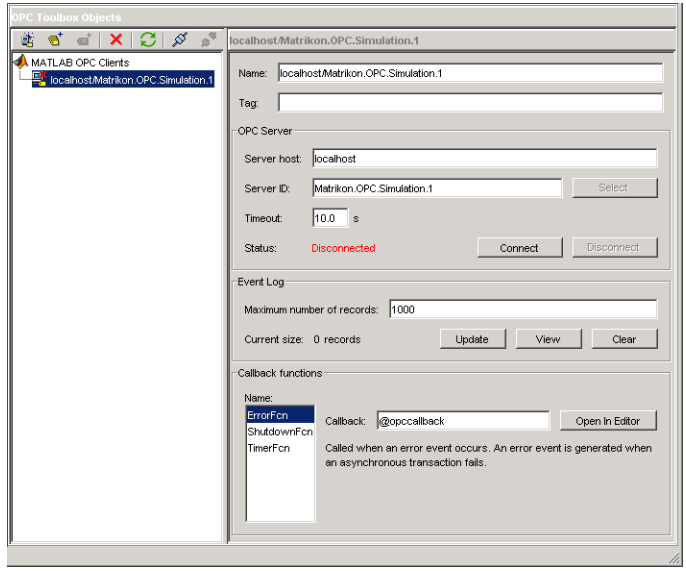
With the OPC Data Access Explorer, you can create a client directly from the **Hosts and OPC Servers** pane.

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.

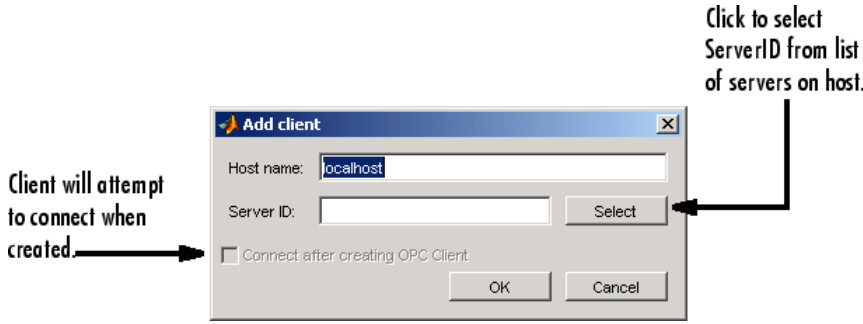


Alternative Methods for Creating Clients

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 the methods described above, a dialog appears requesting the hostname and server ID.



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 Toolbox Data Access Objects” on page 6-2.

Step 4: Connect to the OPC Server

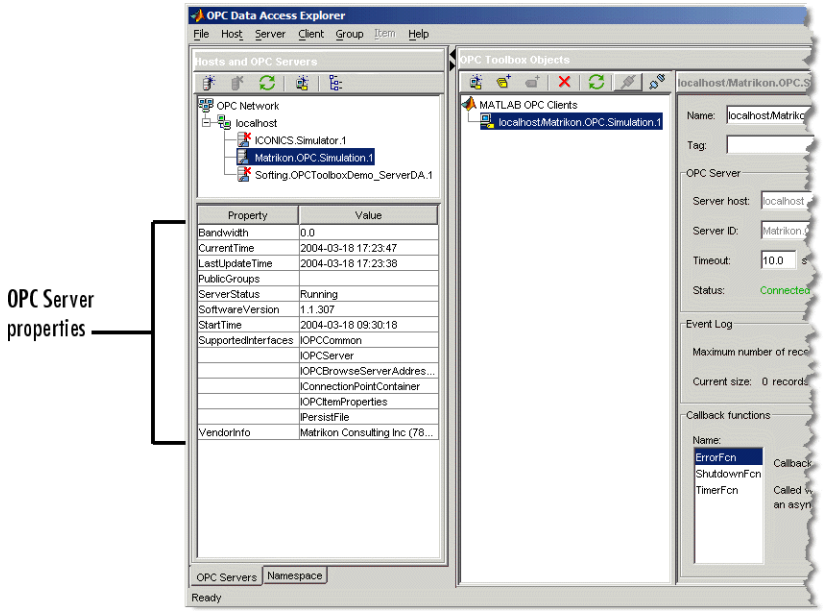
OPC Data Access Client objects are not automatically connected to the server when they are created. This allows you to fully configure an OPC Toolbox object hierarchy (a client with groups and items) prior to connecting to the server, or without a server even being present.

Note The Add Client dialog described in "Alternative Methods for Creating Clients" on page 3-7 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" on page 1-21 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.



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

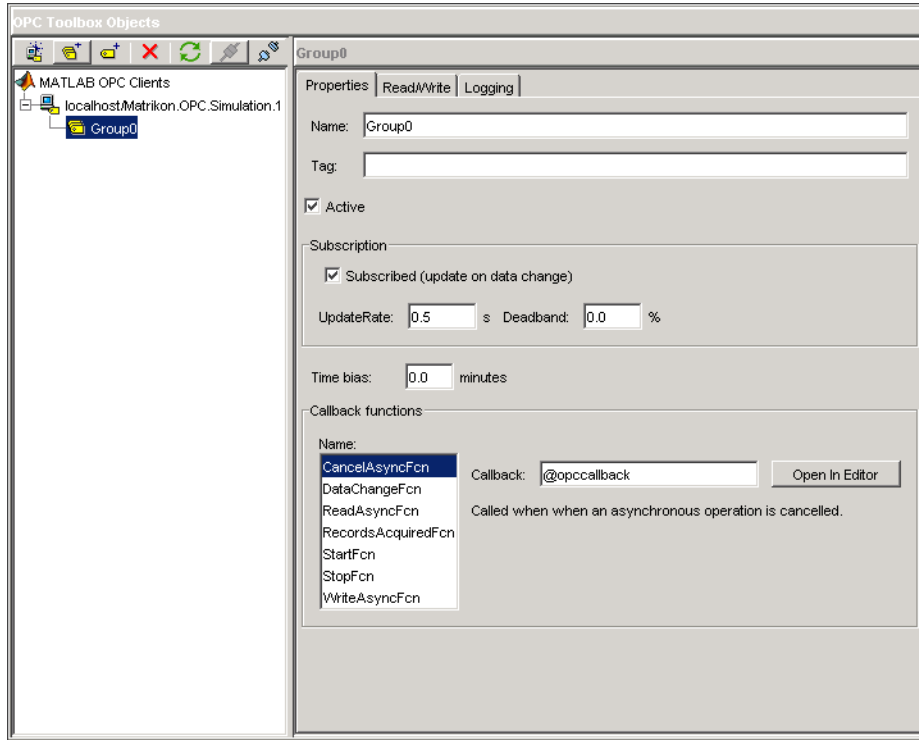
You use the `connect` function to connect an `opcda` client object to the server at the command line.

```
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 OPC Toolbox software.



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 6-5 for more information on creating group objects from the command line.

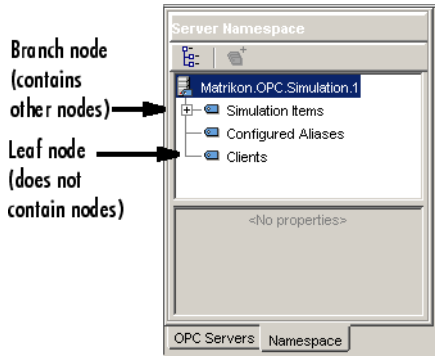
Step 6: Browse the Server Name Space

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 3-15. 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.

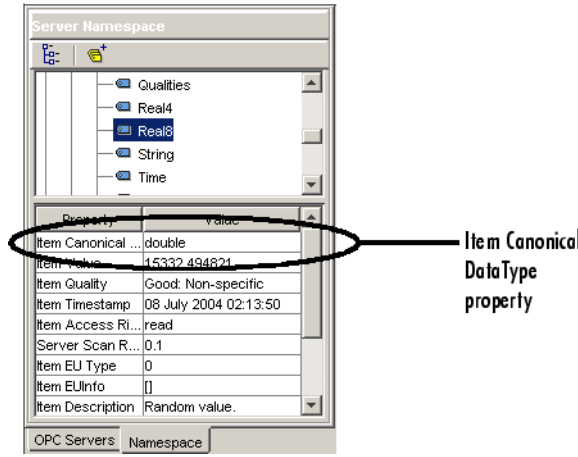


Note If you choose to view the name space as flat, you get a single list of all server items in the name space, expanded to their fully qualified names. A fully qualified name can be used to create a `daitem` object.

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. You will see the properties of the server item in the properties table below the name space tree, as shown in the following figure.



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 will 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 string to limit the returned results. The string 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.

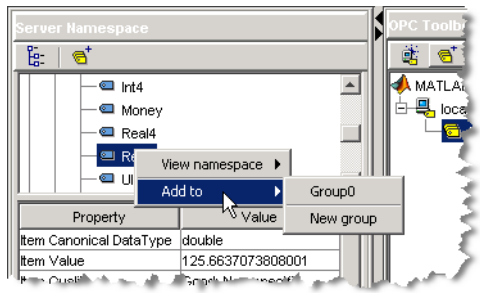
Step 7: Add OPC Data Access Items to the Group

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.

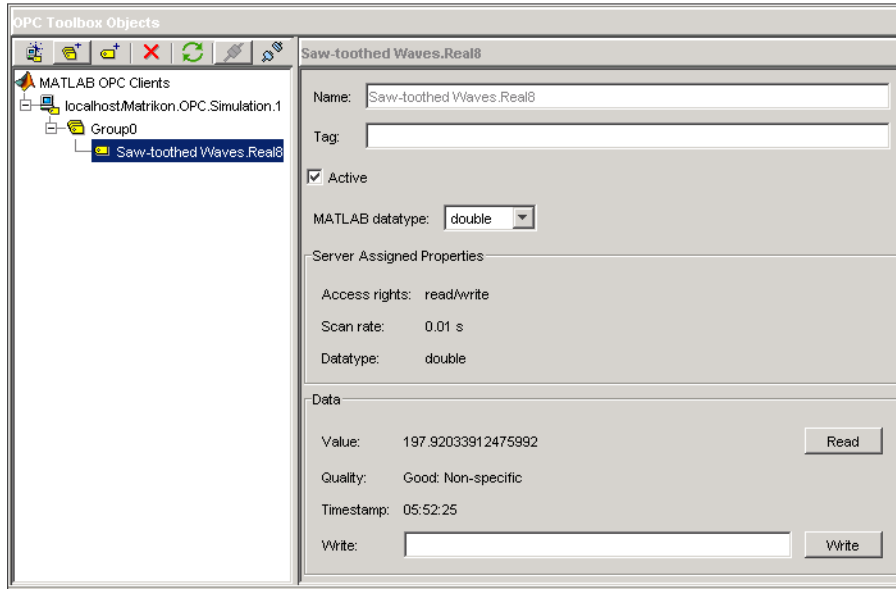
Using the app, you create items directly from the name space tree, using a context menu on each node in the tree.

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.



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.



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 string. 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 })
```

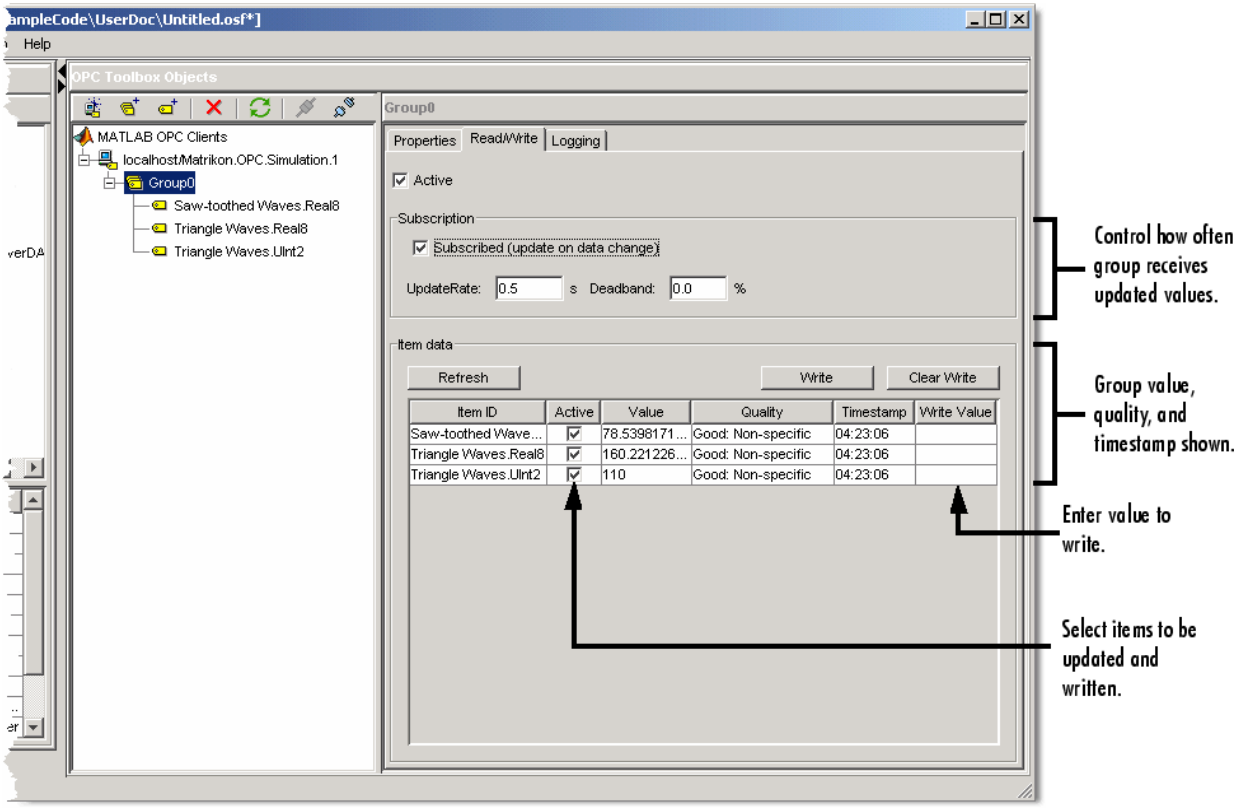
```
itms =
   OPC Item Object Array:
   Index:  DataType:  Active:  ItemID:
   1       double     on       Triangle Waves.Real8
   2       uint16     on       Triangle Waves.UInt2
```

For more information on adding items to groups, see “Create Data Access Item Objects” on page 6-7.

Step 8: View All Item Values

You can view the Value, Quality, and Timestamp for each item using the item's 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's 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.



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 7-11.

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 page 7-2.

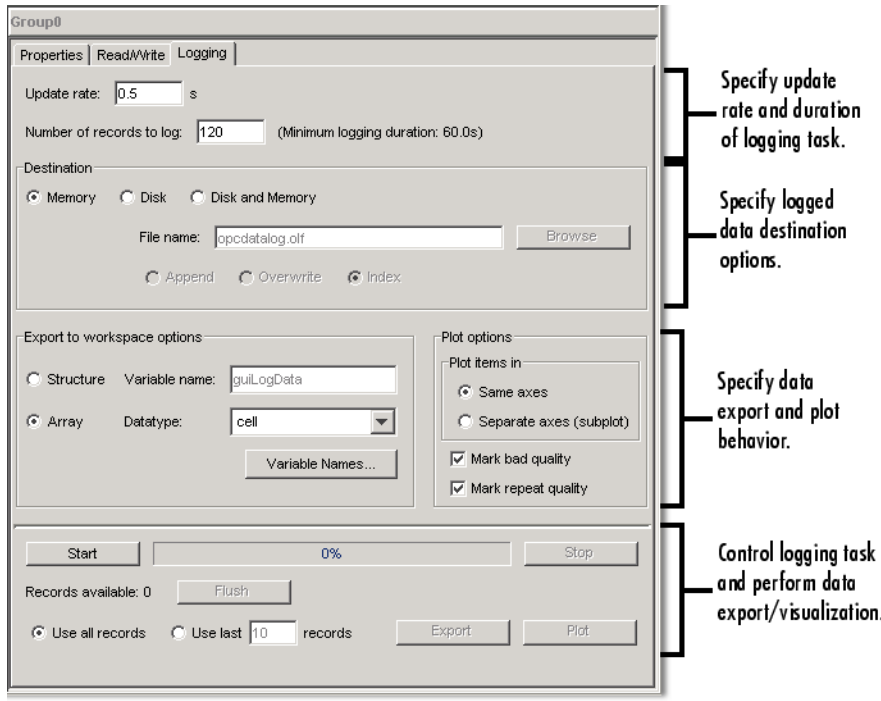
Step 9: Configure Group Properties for Logging

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. OPC Toolbox software allows you to log all 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.



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

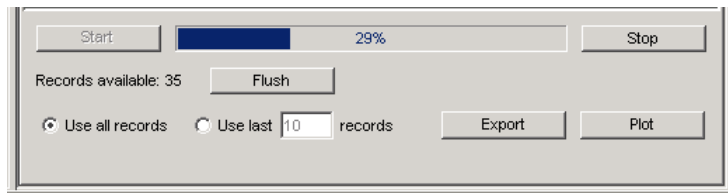
You use the `set` function to set OPC Toolbox object properties. From the MATLAB command line, you can calculate the number of records required for the logging task.

```
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's 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 will begin, and the OPC Toolbox software engine will receive and store 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)
```

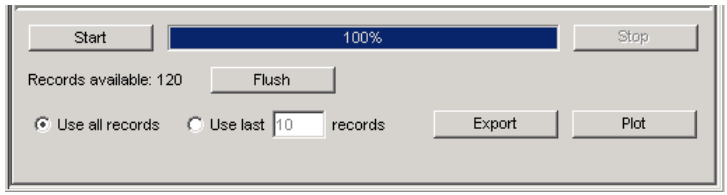
Although the logging operation takes place in the background, you can instruct MATLAB to wait for the logging task to complete, using the `wait` function.

```
wait(grp)
```

Step 11: Plot the Data

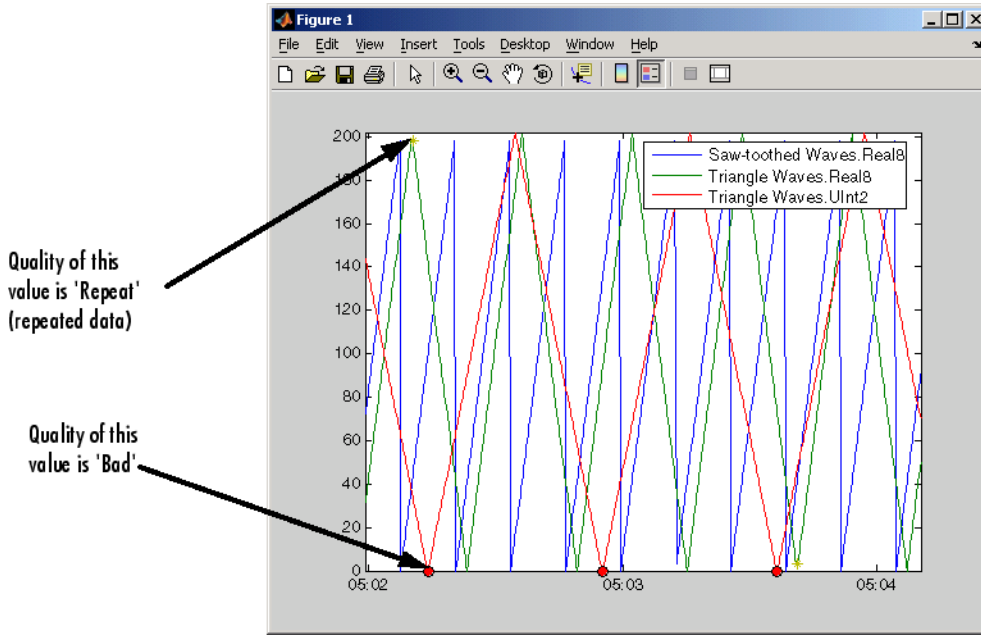
In this introductory example, you use the app to visualize the data logged in Step 10. In a more complex task, you would export the logged data to the workspace and use MATLAB functions to analyze and interpret the logged data.

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 Toolbox objects from the MATLAB command line, or from your own functions, you must remove them from the OPC Toolbox 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 Toolbox object management, see “Delete Objects” on page 6-24.

Quick Start: Using OPC Historical Data Access Functions

The best way to learn about OPC Toolbox 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

This section illustrates the basic steps to create an OPC Toolbox 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 the Matrikon OPC Simulation Server” on page 1-19. 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 Toolbox 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 string, 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 Toolbox objects at the MATLAB command line or from your own functions, you must remove them from the OPC 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 Toolbox object management are discussed in “Delete Objects” on page 6-24.

Data Access User s Guide

5

Introduction to OPC Data Access (DA)

Discover Available Data Access Servers

In this section...

Prerequisites

To interact with an OPC server, OPC Toolbox software needs 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`	Text string 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

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 the Matrikon OPC Simulation Server" on page 1-19. 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 =
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
```

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 Toolbox Data Access Object Properties" on page 6-18.
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 6-5.

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" on page 1-21.

Once 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. See `opcserverinfo` in the function reference for more information.
- Browse the OPC server name space for information on the available server items. See "Browse the OPC DA Server Name Space" on page 5-6 for details on browsing the server name space.
- Create group and item objects to interact with OPC server data. See "Create OPC Toolbox Data Access Objects" on page 6-2 for information on creating group and item objects.

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 5-8 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 string.
`FullyQualifiedID`	The fully qualified item ID of the node, as a string. 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 with OPC Toolbox software.

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-5 (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-7 (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 a server item's properties 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 to you.

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 strings. See the function reference for more information on the `serveritems` function.

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 Toolbox 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 Toolbox Data Access objects are described in more detail in "Toolbox Object Hierarchy for the Data Access Standard" on page 6-2.

Create OPC Toolbox Data Access Objects

In this section...

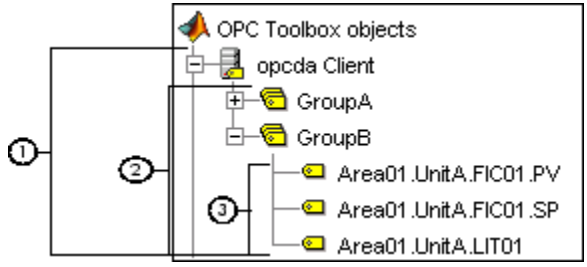
Overview to Objects

The first step in interacting with an OPC server from MATLAB software is to establish a connection between the server and OPC Toolbox software. 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 Toolbox DA software 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.

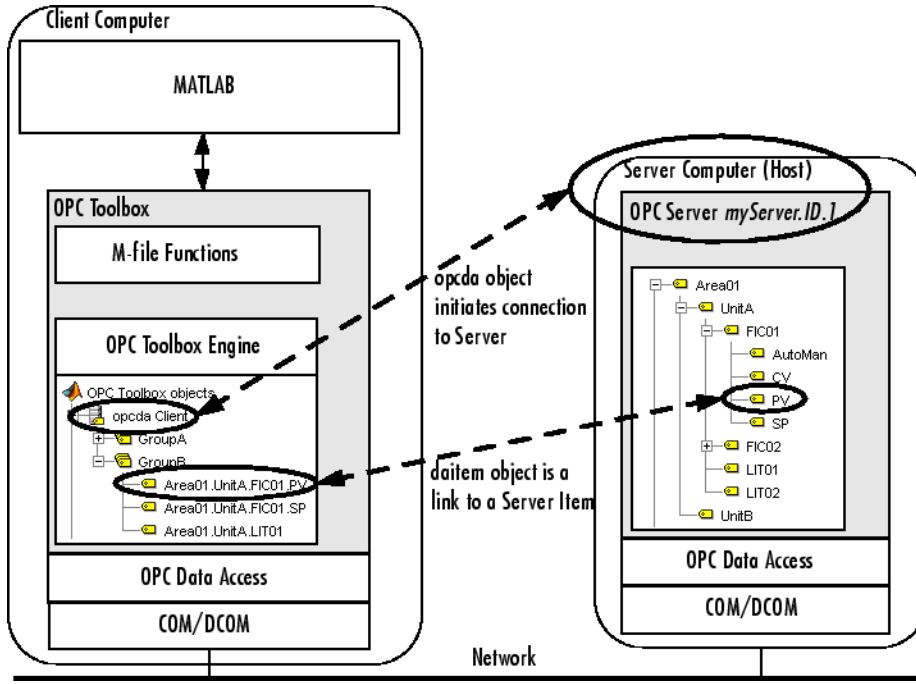


1. **OPC Data Access Client objects** (`opcda` client objects) represent a specific OPC client instance that can communicate with only one server. You define the server using the `Host` and `ServerID` properties. The `Host` property defines the computer on which the server is installed. The `ServerID` 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 Objects Relate to OPC DA Servers

OPC Toolbox software 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 Toolbox Data Access objects and an OPC server.



The `opcda` client object establishes the connection between OPC Toolbox software 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 6-5 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

It is required that each group created under a specific client object has 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 string.

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 6-24 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 7-11.
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 7-11 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 7-15 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 7-15 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 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 7-11 for more information.
- Log data from all items in that group, for later processing and analysis. “Log OPC Server Data” on page 7-15 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 6-5 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 string 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 8-16 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 6-5. 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 8-16 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 string 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 =
(1) Summary of OPC Data Access Item Object: Random.Real8
(2)    Object Parameters
          Parent        : Group0
          Access Rights : read
(3)    Object Status
          Active        : on
(4)    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 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 7-15 for information on logging data.

Build an Object Hierarchy with a Disconnected Client

When you create objects with a connected client, OPC Toolbox software 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 may wish to build up a toolbox object hierarchy without connecting to the server. For example, you may be off site and wish 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.

OPC Toolbox software allows you to configure an entire toolbox 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, OPC Toolbox software will perform some basic validation on the objects you create, and revalidate those objects with the server when you connect to the server.

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, OPC Toolbox software automatically assigns a unique name `groupN`, 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 will be generated if a group with the same name already exists.

- When adding an item to a group using the `additem` function, an error will be 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 may 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 may not be valid for that server, or a group name may 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 Toolbox Data Access Object Vectors

OPC Toolbox software supports the use of *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's `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, OPC Toolbox software 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.
- OPC Toolbox software 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 6-12 describes how to create object vectors.

- "Display a Summary of Object Vectors" on page 6-13 describes how object vectors are displayed at the command line.
- "Use Object Vectors" on page 6-13 describes how you can use object vectors with OPC Toolbox software.

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 Toolbox Data Access object vector using concatenation, you 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 6-20.

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 Toolbox 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 OPC Toolbox 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 will be 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`.

Using OPC Toolbox software, 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 using OPC Toolbox software:

- “Define a New Public Group” on page 6-15 describes how you can create new public groups.
- “Connect to an Existing Public Group” on page 6-15 describes how you can utilise a public group that is already defined on the server.

- "Remove Public Groups from the Server" on page 6-16 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 6-5) 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 Toolbox Data Access Object Properties

In this section...

Purpose of Object Properties

All OPC Toolbox 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 OPC Toolbox software.

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 Toolbox 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. You 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 Toolbox 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` functions.

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 Toolbox 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 the value of a property by assigning the value to the object as if it were a MATLAB structure.

```
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: string -or- function handle -or- cell array
    DataChangeFcn: string -or- function handle -or- cell array
    ReadAsyncFcn: string -or- function handle -or- cell array
    RecordsAcquiredFcn: string -or- function handle -or- cell array
    RecordsAcquiredFcnCount
    StartFcn: string -or- function handle -or- cell array
    StopFcn: string -or- function handle -or- cell array
    WriteAsyncFcn: string -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 Toolbox 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 7-15.
- `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 6-14.

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.

Delete Objects

When you finish using your OPC Toolbox 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 Toolbox 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.

Save and Load Objects

Using the `save` command, you can save an OPC Toolbox 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 Toolbox 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.

7

Reading, Writing, and Logging OPC Data

The core of any OPC Toolbox software 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.

Using OPC Toolbox software 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 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

In this section...

Introduction to Reading and Writing

Using OPC Toolbox software, you can exchange data with the OPC 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 will wait 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 will wait 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 asyncrhonous 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 7-7 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. OPC Toolbox software 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 8-16.

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. OPC Toolbox software 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 Toolbox Data Access Object Vectors” on page 6-11 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 7-8 describes how to read multiple values from an item vector or `dagroup` object.
- "Write Multiple Values" on page 7-9 describes how to write multiple values to an item vector or `dagroup` object.
- "Error Handling for Multiple Item Read and Write Operations" on page 7-9 explains how OPC Toolbox software deals with 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 9-5.

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. OPC Toolbox software requires these multiple values to be in a *cell array*, since the data types for each value may be different. For information on constructing cell arrays, see MATLAB Programming.

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

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*. OPC Toolbox software 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 Toolbox 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 OPC Toolbox software. 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 7-12 describes how to turn off data change notification for a `dagroup` object.
- "Temporarily Disable Items in a Group" on page 7-13 describes how to control which items in a group must be monitored for data changes.

- "Customize the Data Change Event Response" on page 7-14 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 OPC Toolbox Software Processes Data Change Events

OPC 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 7-15.

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 OPC 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 OPC Toolbox Software Logs Data

The OPC Data Access Specification, which OPC Toolbox software implements, 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. OPC 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 OPC 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.

OPC 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 7-11.

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 7-13). 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 Toolbox object hierarchy

This example creates a hierarchy of OPC Toolbox 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 7-18 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 7-19 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 7-21 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 7-21 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 7-23 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 7-18). 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 7-19). 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 7-20). 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 OPC Toolbox Software Logs Data" on page 7-15), 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 OPC Toolbox software 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

OPC 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 OPC Toolbox software 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 string 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 OPC Toolbox software 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`, OPC Toolbox software 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 9-18.

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 OPC Toolbox software 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 OPC Toolbox software 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 7-23 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

OPC 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 7-23, discusses how to retrieve data from the toolbox engine into MATLAB.
- "Retrieve Data from Disk" on page 7-25, 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 OPC Toolbox software 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 string 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 7-16.

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 string, that should be used for the returned values. Valid data type strings 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 strings, 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 7-16, 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 OPC Toolbox software handles data returned by an OPC server.

OPC Data: Value, Quality, and TimeStamp

In this section...

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 string 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 the `Quality` string, see "OPC Quality Strings" 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.

OPC Toolbox software provides access to the `Value`, `Quality`, and `TimeStamp` properties of a server item through properties of the `daitem` object associated with that server item.

How Value, Quality, and TimeStamp Are Obtained

OPC Toolbox software 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 way OPC Toolbox software uses the three OPC Data Access mechanisms is described in the following sections:

- "OPC Data Returned from Synchronous Read Operations" on page 8-4 describes the synchronous read mechanism used by the `read` function.
- "OPC Data Returned in Asynchronous Read Operations" on page 8-4 describes the asynchronous read mechanism used by the `readasync` function.
- "OPC Data Returned from a Data Change Event" on page 8-5 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.

OPC Toolbox software 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.

OPC Toolbox software 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 9-9.

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, OPC Toolbox software 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 9-15.
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 7-15.
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 9-15.

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 9-9.

Work with Structure-Formatted Data

In this section...

When Structures Are Used

OPC 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 Toolbox Group Objects

This example creates a hierarchy of OPC Toolbox 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

OPC 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 8-13.

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 9-16.

Array-Formatted Data

In this section...

Array Content

OPC 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 8-14 for information on how the `Value` array is populated.
- `Quality` — An *`nRecs`*-by-*`nItems`* cell array of quality strings. 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 OPC Toolbox software (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. OPC Toolbox software 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 8-14 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 8-14 discusses how the toolbox deals with this issue.

Treatment of Missing Data

When OPC Toolbox software 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 string `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, OPC Toolbox software 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

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 OPC 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`	

MATLAB Data Type	OPC Server Data Type (COM Variant Type)	Remarks
`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. OPC Toolbox software converts the value into the equivalent COM Variant according to Table 8-1. 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 OPC Toolbox software receives the value, the COM Variant is converted to the equivalent MATLAB data type according to Table 8-1.

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. OPC 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 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 Toolbox 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

In this section...

Overview to Callback Example

To illustrate how to use callbacks, this section presents a simple example that creates an OPC Toolbox 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 9-5.

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 9-15.

Step 1: Create OPC Toolbox Group Objects

This example creates a hierarchy of OPC Toolbox 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

OPC 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 9-9.

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 7-11.

Event	Callback Property	Description
		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.
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.

Event	Callback Property	Description
		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.

Event	Callback Property	Description
		When a stop event occurs, the toolbox executes the function specified by the `StopFcn` property. Stop events occur at the `dagroup` object level.
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

Each event has a set of information associated with that event. The information is generated by the OPC server or the OPC 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 9-15.

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 text string that specifies the event type. For a start event, this field contains the text string `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 9-9
- "Data Fields for Start, Stop, and Records Acquired Events" on page 9-10
- "Data Fields for Shutdown Events" on page 9-11
- "Data Fields for Timer Events" on page 9-12

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.
`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 string.
`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 string 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.

Field Name	Description
`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 string 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 9-9.

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 9-15.

To illustrate the event log, this example creates an OPC Toolbox object hierarchy, executes a logging task, and then examines the object's `EventLog` property:

Step 1: Create the OPC Toolbox Object Hierarchy

This example creates a hierarchy of OPC Toolbox 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

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 Toolbox 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 9-9.

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 9-17.

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 OPC Toolbox software 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 9-18.

Specify Callback Functions

You associate a callback function with a specific event by setting the value of the OPC Toolbox 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 Text String to Specify Callback Functions" on page 9-17
- "Use a Cell Array to Specify Callback Functions" on page 9-17
- "Use Function Handles to Specify Callback Functions" on page 9-18

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 Text String to Specify Callback Functions

You can specify the callback function as a string. 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 text string 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 Toolbox 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 Toolbox object hierarchy and sets the records acquired event callback function property to the `display_opcdata` function, created in “Write a Callback Function” on page 9-16.

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 Toolbox Object Hierarchy

This example creates a hierarchy of OPC Toolbox 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 9-16. 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 Toolbox Block Library

Block Library Overview

OPC Toolbox software includes a Simulink interface called the OPC Toolbox 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 Toolbox block library with blocks from other Simulink libraries to create models capable of sophisticated OPC server communications.

The OPC Toolbox 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 Toolbox block library is to observe an example, such as the one provided in “Read and Write Data from a Model” on page 10-3.

Read and Write Data from a Model

In this section...

Example Overview

This section provides a step-by-step example to illustrate how to use the OPC Toolbox block library. The example builds a simple model using the blocks in the OPC Toolbox 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 the Matrikon OPC Simulation Server" on page 1-19. 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
```

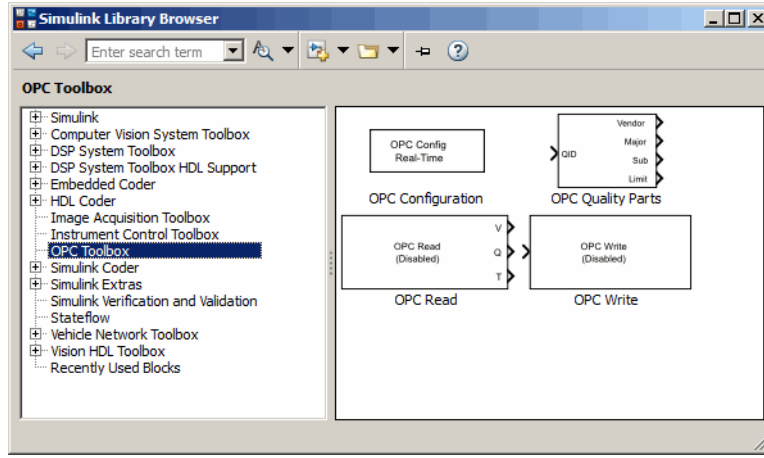
In the Simulink start page dialog, click **Blank Model**, and then **Create Model**. An empty, Editor window opens.

2 In the Editor, click **File** > **Save As** to assign a name to your new model.

Step 2: Open the OPC Toolbox Block Library

1 In the model Editor window, click **View** > **Library Browser**.

2 The Simulink Library Browser opens. Its left pane contains a tree of available block libraries in alphabetical order. Click the `OPC Toolbox` node.



Alternatively, you can open the OPC Toolbox block library by typing the following command at the MATLAB command prompt:

```
opclib
```

Step 3: Drag OPC Toolbox Blocks into the Editor

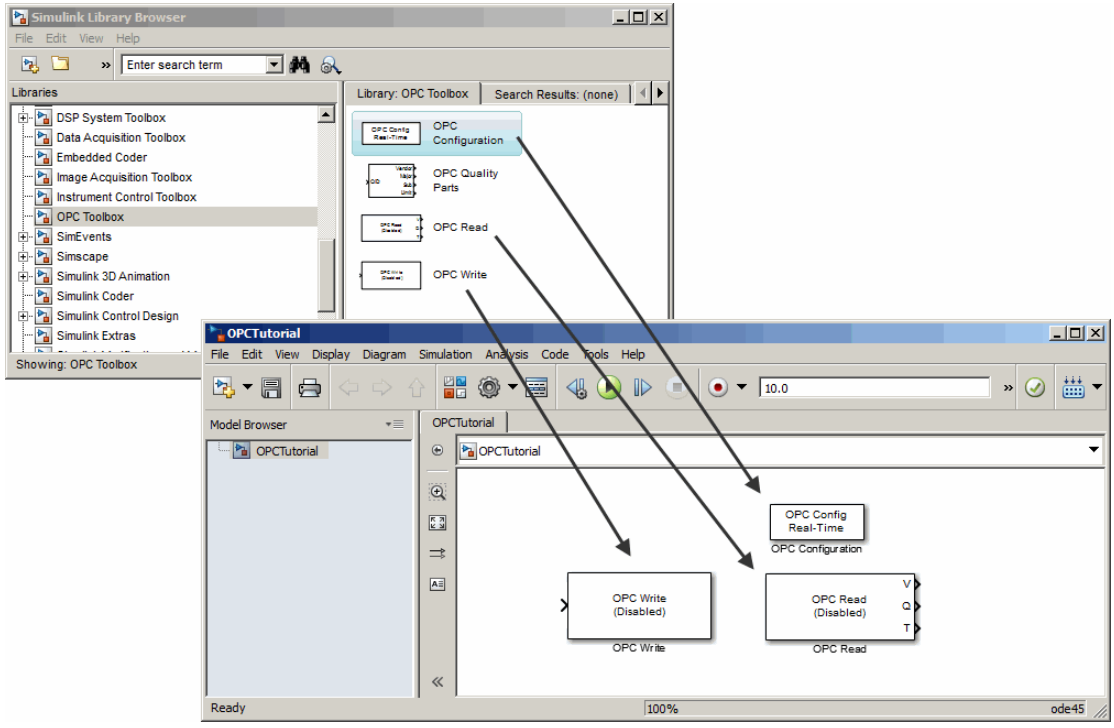
The OPC Toolbox block library contains four blocks

- `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.

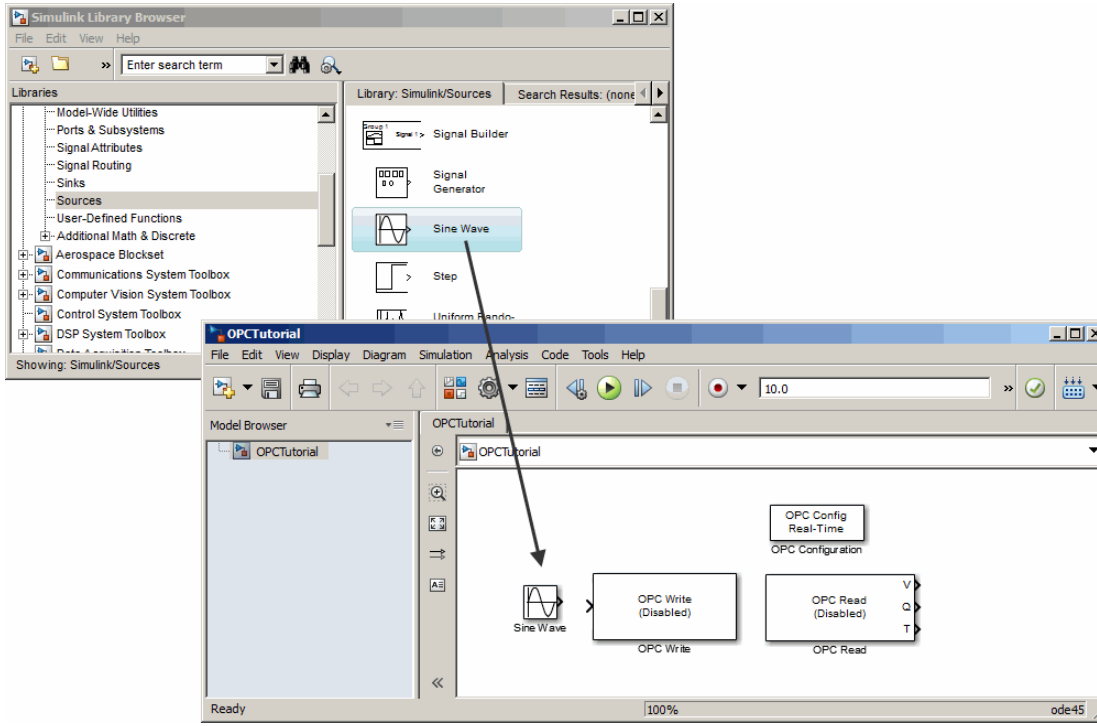
To use the blocks in a model, select each block in the library and, holding the mouse button down, drag the block into the Simulink Editor. For this example, you need one instance each of the OPC Configuration, OPC Write, and OPC Read block in your model.



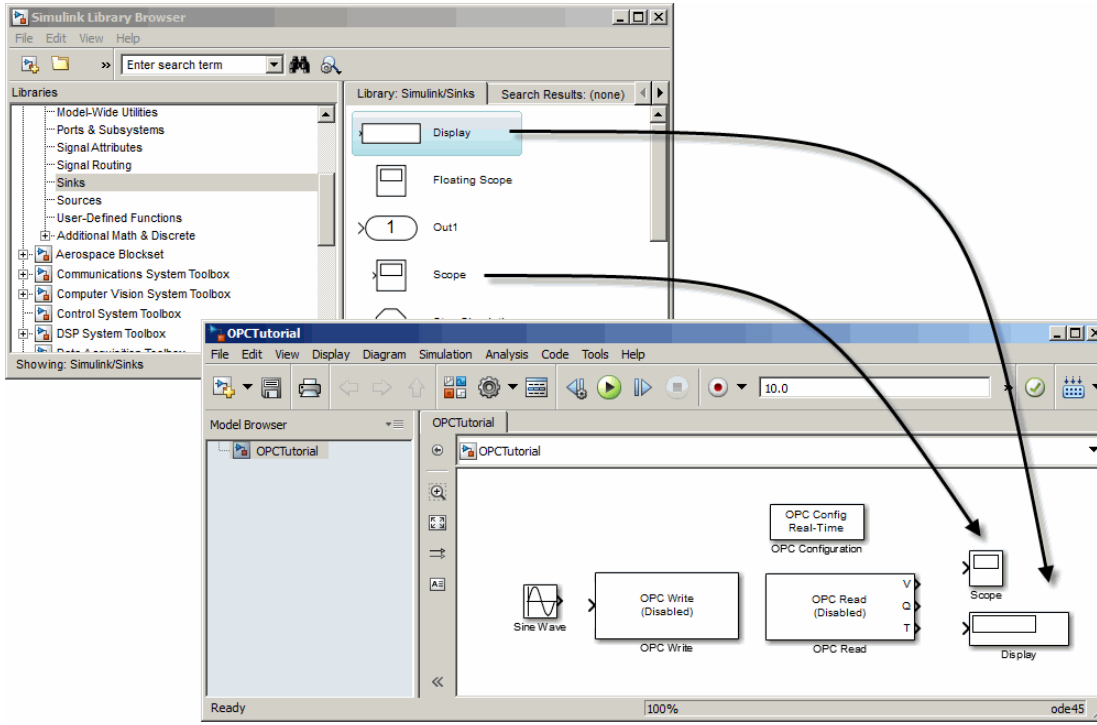
Step 4: Drag Other Blocks to Complete the Model

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.



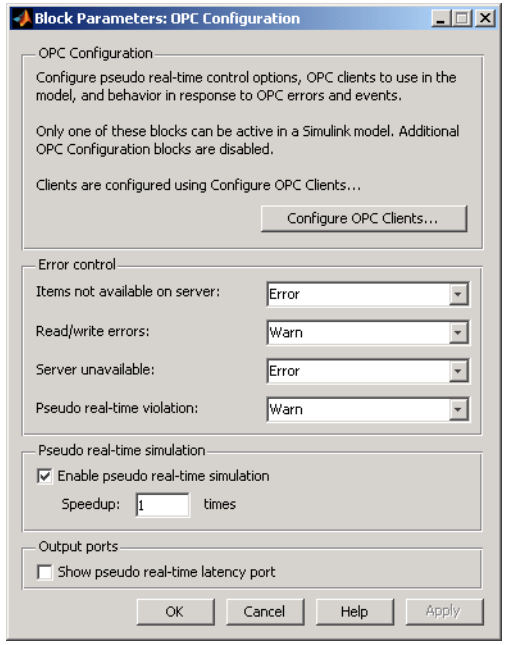
You can use the Scope block to show the value received from the server, and a Display block to view the quality of the item. (You will remove the time stamp output port in the next step.) To access the Scope block, click the Sinks library entry in the expanded Simulink node in the browser tree. From 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.


Simulink Library Browser
File Edit View Help
Enter search term
Libraries
Model-Wide Utilities
Ports & Subsystems
Signal Attributes
Signal Routing
Sinks
Sources
User-Defined Functions
Additional Math & Discrete
Aerospace Blockset
Communications System Toolbox
Computer Vision System Toolbox
Control System Toolbox
DSP System Toolbox
Showing: Simulink/Sinks
Library: Simulink/Sinks
Search Results: (none)
Display
Floating Scope
Out1
Scope
OPCTutorial
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
Model Browser
OPCTutorial
OPC Config Real-Time
OPC Configuration
Sine Wave
OPC Write (Disabled)
OPC Write
OPC Read (Disabled)
OPC Read
V
Q
T
Scope
Display
Ready
100%
ode45

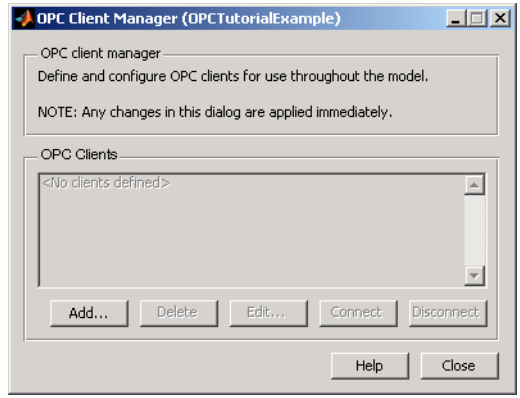
Step 5: Configure OPC Servers for the Model

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.

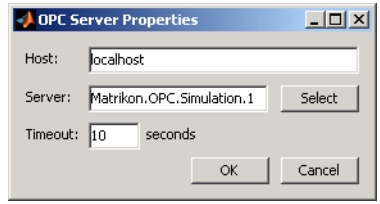
1 Double-click the OPC Configuration block to open its parameters dialog.



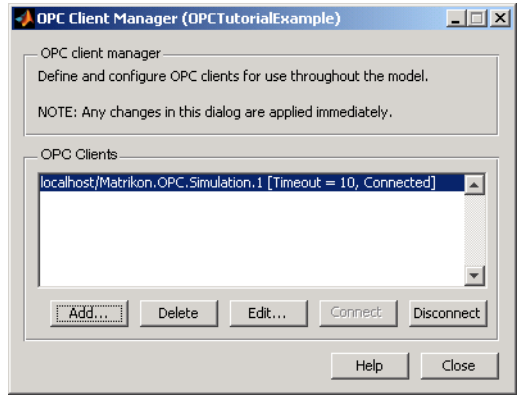
2 Click **Configure OPC Clients** to open the OPC Client Manager.



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).



4 Click **OK** to add the OPC server to the OPC Client Manager.



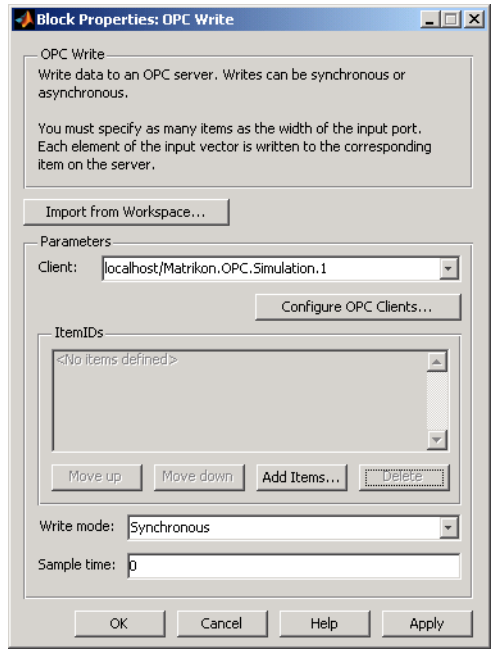
The Matrikon OPC Simulation Server is now available throughout the model for reading and writing.

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.

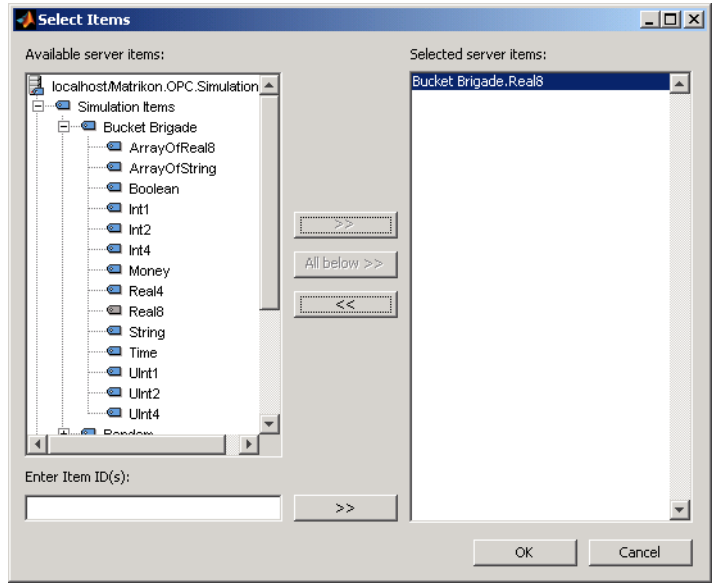
Step 6: Specify the Block Parameter Values

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.



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.



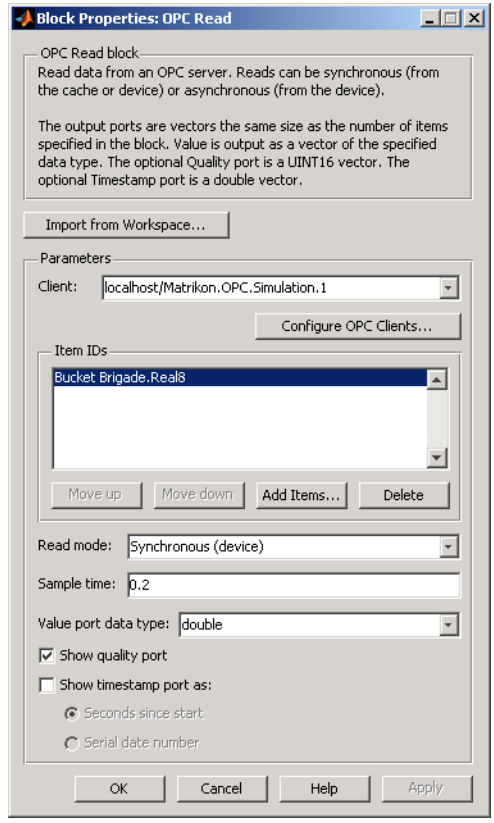
4 Click **OK** to add the item `Bucket Brigade.Real8` to the OPC Write block's ItemIDs list.

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.

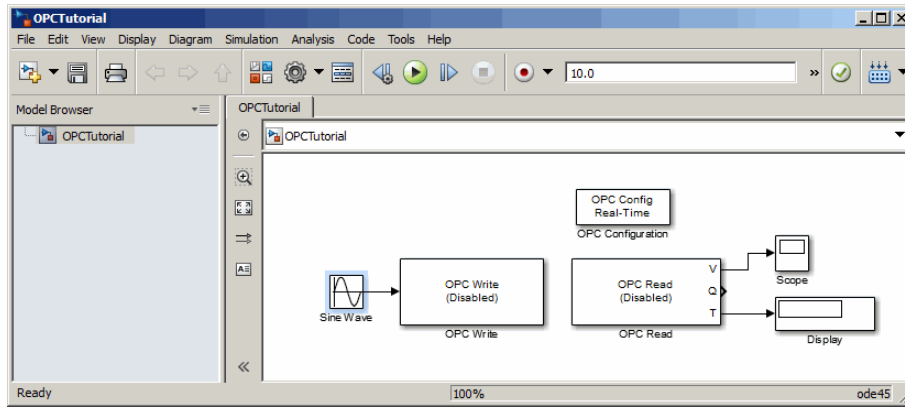

Block Properties: OPC Read
OPC Read block
Read data from an OPC server. Reads can be synchronous (from the cache or device) or asynchronous (from the device).
The output ports are vectors the same size as the number of items specified in the block. Value is output as a vector of the specified data type. 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.

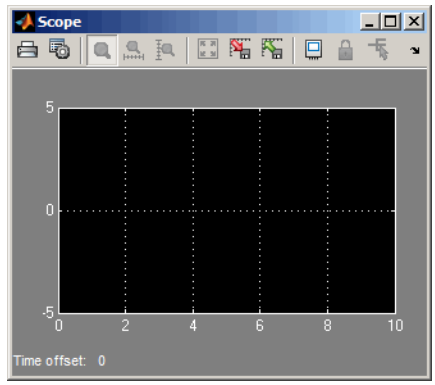
In the same way, make a connection between the first output port of the OPC Read block (labeled `V`) and the input port of the Scope block. Then connect the other output port of the OPC Read block (labeled `Q`) to the input port of the Display block.

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 10-8.

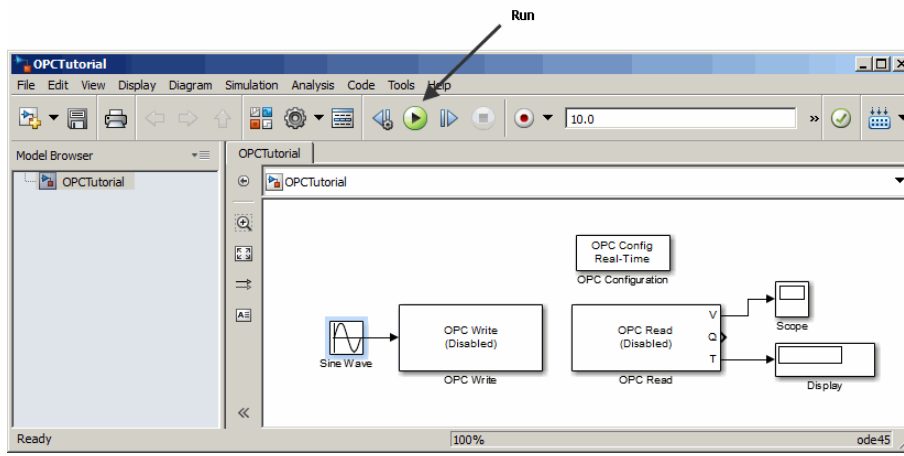


Step 8: Run the Simulation

Before you run the simulation, double-click the Scope block to open the scope view.

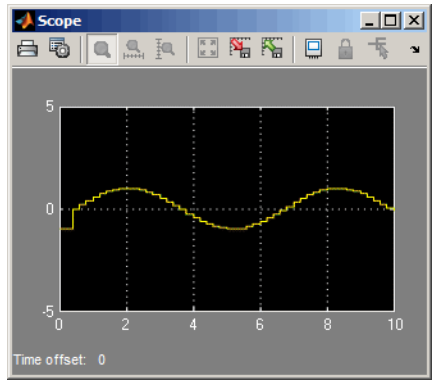


To run the simulation, click **Run** in the Simulink Editor toolstrip. Alternatively, you can select the Simulink Editor menu option **Simulation > Run**.



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 Strings” 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.



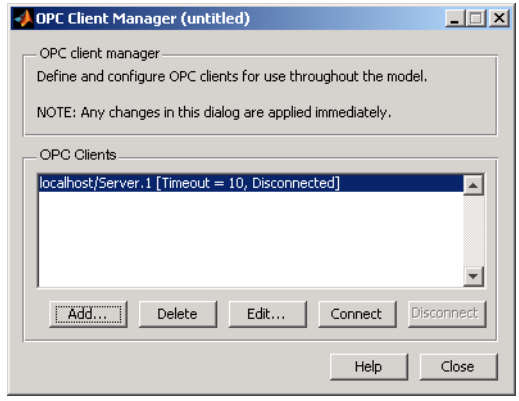
Use the OPC Client Manager

In this section...

Introduction to the OPC Client Manager

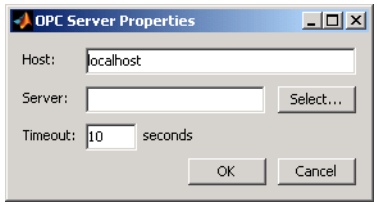
The OPC Client Manager displays and manages all clients for a Simulink model. Using the OPC Client Manager, you associate one or more clients with a particular model. Each time you use an OPC Read or OPC Write block, you choose the client for that block from the list of configured clients. By defining 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.



Add Clients to the OPC Client Manager

You add clients to the OPC Client Manager by clicking **Add**. The following dialog box appears.



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.

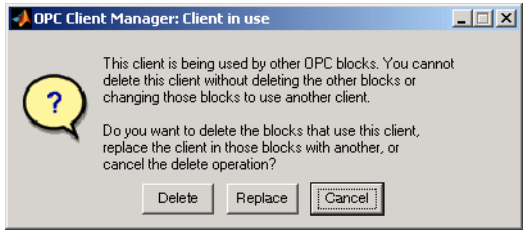
Specify the timeout (in seconds) to use when communicating with the server.

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 Toolbox library blocks, you see the following dialog.



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

OPC 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's connection status by highlighting a client in the **OPC Client** list and clicking **Connect** or **Disconnect**.

The OPC Toolbox block library automatically reconnects any disconnected client to its server when you run a simulation.

Properties — Alphabetical List

AccessRights

Inherent nature of access to item

Description

`AccessRights` represents the server's ability to access a single OPC data item. The property value can be `read`, `write`, or `read/write`. If `AccessRights` is `read`, you can read the server item's value. If `AccessRights` is `write`, you can write values to the server item. If `AccessRights` is `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 may return an error.

Characteristics

Access	Read-only
Applies to	`daitem`
Data type	string
Values	`[read	read/write	write]` The value is set by the server when an item is created.

See Also

Functions

`read`, `readasync`, `refresh`, `write`, `writeasync`

Properties

`Subscription`

Active

Group or item activation state

Description

`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 a `dagroup` or `daitem` 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.

Characteristics

Access	Read/write
Applies to	`dagroup`, `daitem`
Data type	`string`
Values	`[off	{ on }]`

See Also

Functions

`read`, `readasync`, `refresh`

Properties

`DataChangeFcn`, `Subscription`

CancelAsyncFcn

Callback function file to execute when asynchronous operation is canceled

Description

You configure `CancelAsyncFcn` to execute a callback function file 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.

Characteristics

Access	Read/write
Applies to	`dagroup`
Data type	String, function handle, or cell array
Values	The default value is `@opccallback`.

See Also

Functions

`cancelasync`, `opccallback`, `readasync`, `writeasync`

Properties

EventLog

CanonicalDataType

Server's data type for item

Description

`CanonicalDataType` indicates the data type of the item as stored on the OPC server. The MATLAB supported data types are as for 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.

Characteristics

Access	Read-only
Applies to	`daitem`
Data type	string
Values	The default value is determined when the item is created.

See Also

Functions

`additem`

Properties

`DataType`

DataChangeFcn

Callback function file to execute when data change event occurs

Description

You configure `DataChangeFcn` to execute a callback function file 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

Characteristics

Access	Read/write
Applies to	`dagroup`
Data type	string, function handle, or cell array
Values	The default value is an empty matrix ([ ]).

See Also

Functions

`opccallback`, `refresh`

Properties

`Active`, `DeadbandPercent`, `Subscription`, `UpdateRate`

DataType

Client item's data type

Description

`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 `unknown` 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 Toolbox 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

OPC Toolbox Data Type	COM Data Type	MATLAB Data Type
currency	VT_CY	double
date	VT_DATE	double
logical	VT_BOOL	logical
double	VT_EMPTY	Empty array ([ ])

Characteristics

Access	Read-only while logging
Applies to	daitem
Data type	string
Values	[{ unknown } | double | char | single | uint8 | uint16 | uint32 | uint64 | int8 | int16 | int32 | int64 | currency | date | logical]

See Also

Functions

additem

Properties

CanonicalDataType

DeadbandPercent

Percentage change in item value that causes subscription callback

Description

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`

The `DeadbandPercent` property only affects items that have an analogue 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 may 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 will be 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.

Characteristics

Access	Read/write
Applies to	`dagroup`
Data type	double
Values	Any value from `0` to `100`, inclusive. The default value is `0`.

See Also

Functions

`serveritemprops`

Properties

`Active`, `Subscription`, `UpdateRate`

ErrorFcn

Callback function file to execute when error event occurs

Description

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.

Characteristics

Access	Read/write

Applies to	`opcda`
Data type	`string`, function handle, or cell array
Values	`@opccallback` is the default callback function.

See Also

Functions

`opccallback`, `showopcevents`

Properties

`EventLog`, `Timeout`

EventLog

Event information log

Description

`EventLog` contains a structure array that stores information related to OPC 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 `clearеventlog` function.

Characteristics

Access	Read-only
Applies to	`opcda`
Data type	Structure array
Values	The default value is an empty matrix (`[]`).

Examples

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]
```

See Also

Functions

`cleareventlog`, `start`

Properties

`CancelAsyncFcn`, `DataChangeFcn`, `EventLogMax`, `ErrorFcn`, `ReadAsyncFcn`, `StartFcn`, `StopFcn`, `WriteAsyncFcn`

EventLogMax

Maximum number of events to store in event log

Description

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`.

Characteristics

Access	Read/write
Applies to	`opcda`
Data type	double
Values	Any integer in the range `[0 Inf]`. The default value is `1000`.

See Also

Functions

`cleareventlog`

Properties

`EventLog`

Group

Data Access Group objects contained by client

Description

`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.

Characteristics

Access	Read-only
Applies to	`opcda`
Data type	`dagroup` array
Values	The default value is an empty array (`[ ]`).

See Also

Functions

`addgroup`, `delete`

GroupType

Public status of `dagroup` object

Description

`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 any other OPC client on the network.

Characteristics

Access	Read-only
Applies to	`dagroup`
Data type	string
Values	`[ { private } |  public  ]`

See Also

Functions

`addgroup`

Host

DNS name or IP address of server

Description

`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.

Characteristics

Access	Read-only while connected
Applies to	`opcda`
Data type	string
Values	The value is configured when the object is created.

See Also

Functions

`opcda`

Properties

`ServerID`

Item

Data Access Item objects contained by group

Description

`Item` is a vector of `daitem` objects contained by 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.

Characteristics

Access	Read-only
Applies to	`dagroup`
Data type	`daitem`
Values	The default value is an empty matrix (`[ ]`).

Example

This example creates a fictitious client, adds a group and two items.

```
da = opcda( localhost ,  Dummy.Server );
grp = addgroup(da,  MyGroup );
itm1 = additem(grp,  Item.Name.1 );
itm2 = additem(grp,  Item.Name.2 );
allItems = grp.Item
```

If one of the items is deleted, the Item property is updated to reflect this.

```
delete(itm2);
newItems = grp.Item
```

See Also

Functions

`additem`, `delete`

ItemID

Fully qualified ID on OPC server

Description

`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.

Characteristics

Access	Read-only while connected
Applies to	`daitem`
Data type	string
Values	The default value is set during creation.

See Also

Functions

`additem`, `getnamespace`, `serveritems`

LogFileName

Name of disk file to which logged data is written

Description

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 may 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
...
```

Characteristics

Access	Read-only while logging
Applies to	`dagroup`
Data type	string
Values	The default value is `opcdatalog.olf`.

See Also

Functions

`start`

Properties

`LoggingMode`, `LogToDiskMode`

Logging

Status of data logging

Description

`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.

Characteristics

Access	Read-only
Applies to	`dagroup`
Data type	string
Values	`[ { off } | on ]`

See Also

Functions

`start`, `stop`, `wait`

Properties

`LoggingMode`, `RecordsToAcquire`

LoggingMode

Specify destination for logged data

Description

`LoggingMode` can be set to `disk`, `memory`, or `disk&memory`. If `LoggingMode` is set to `disk`, `DataChange` events (records) are stored to a disk file as specified by `LogFileName`. If `LoggingMode` is set to `memory`, records are stored to memory (the buffer). If `LoggingMode` is set to `disk&memory`, records are stored to memory and to a disk file. `LoggingMode` defaults to `memory`.

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.

Characteristics

Access	Read-only while logging
Applies to	`dagroup`
Data type	string
Values	`[disk	disk&memory	{ memory }]`

See Also

Functions

`getdata`, `opcread`, `refresh`, `start`, `stop`

Properties

`LogFileName`, `RecordsToAcquire`

LogToDiskMode

Method of disk file handling for logged data

Description

`LogToDiskMode` can be set to `append`, `overwrite` or `index`. If `LogToDiskMode` is set to `append`, then 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. If `LogToDiskMode` is set to `overwrite`, then the log file is overwritten each time `start` is called. If `LogToDiskMode` is set to `index`, then 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 will result in the second object failing during the `start` operation.

Characteristics

Access	Read-only while logging
Applies to	`dagroup`
Data type	string

Values	`[append	{ index }	overwrite]`

See Also

Functions

`start`

Properties

`LogFileName`, `Logging`, `LoggingMode`

Name

Descriptive name for OPC Toolbox object

Description

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 string, and is used for display and identification purposes only.

Characteristics

Access	Read/write
Applies to	`opcda`, `dagroup`
Data type	`string`
Values	The default value is defined at object creation time.

See Also

Functions

`opcda`, `addgroup`

Properties

`Host`, `ItemID`, `ServerID`

Parent

OPC Toolbox object that contains `dagroup` or `daitem` object

Description

For `dagroup` objects, `Parent` indicates the `opcda` object that contains the group. For `daitem` objects, `Parent` indicates the `dagroup` object that contains the `daitem` object.

Characteristics

Access	Read-only
Applies to	`dagroup`, `daitem`
Data type	Type of parent object
Values	The value is defined at object creation time.

See Also

Properties

`Group`, `Item`

Quality

Quality of data value as string

Description

`Quality` indicates the quality of the `daitem` object's `Value` property as a string. You can use the `Quality` property to determine if a value is useful or not.

The `Quality` string is made up of a major quality string, a substatus string, and an optional limit status string, arranged 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.
`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, consult “OPC Quality Strings” 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.

Characteristics

Access	Read-only
Applies to	`daitem`
Data type	string
Values	The default value is `Bad: Out of Service`.

See Also

Functions

`read`, `readasync`, `refresh`

Properties

`QualityID`, `Subscription`, `TimeStamp`, `UpdateRate`, `Value`

QualityID

Quality of data value as 16-bit integer

Description

`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 string equivalent of the four quality fields from the `QualityID` value.

For more information on quality values, see “OPC Quality Strings” on page A-2.

`QualityID` is updated when you perform a read operation using `read` or `readasync`, or when a subscription callback occurs.

Characteristics

Access	Read-only
Applies to	`daitem`
Data type	double
Values	An integer from `0` to `65535`. The default value is `28` (representing the quality `Bad: Out of Service`).

See Also

Functions

`bitand`, `bitshift`, `opcqparts`, `opcqstr`, `read`, `readasync`, `refresh`

Properties

`Quality`, `Value`

ReadAsyncFcn

Callback function file to execute when asynchronous read completes

Description

You configure `ReadAsyncFcn` to execute a callback function file 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.

Characteristics

Access	Read/write

Applies to	`dagroup`
Data type	string, function handle, or cell array
Values	The default value is `@opccallback`.

See Also

Functions

`opccallback`, `readasync`

Properties

`EventLog`

RecordsAcquired

Number of records acquired

Description

`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.

Characteristics

Access	Read-only
Applies to	`dagroup`
Data type	double
Values	The default value is `0`.

See Also

Functions

`start`

Properties

`Logging`, `RecordsAcquiredFcn`, `RecordsAvailable`

RecordsAcquiredFcn

Callback function file to execute when `RecordsAcquired` event occurs

Description

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.

Characteristics

Access	Read/write
Applies to	`dagroup`
Data type	String, function handle, or cell array
Values	The default value is an empty matrix (`[]`).

See Also

Functions

`start`

Properties

`EventLog`, `RecordsAcquired`, `RecordsAcquiredFcnCount`

RecordsAcquiredFcnCount

Number of records to acquire before `RecordsAcquired` event occurs

Description

A records acquired event is generated each time the number of records acquired reaches a multiple of `RecordsAcquiredFcnCount`.

Characteristics

Access	Read-only while logging
Applies to	`dagroup`
Data type	double
Values	Any integer in the range `[0 Inf]`. The default value is `20`.

See Also

Properties

`RecordsAcquired`, `RecordsAcquiredFcn`

RecordsAvailable

Number of records available in OPC Toolbox engine

Description

`RecordsAvailable` indicates the number of records that are available in the OPC 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.

Characteristics

Access	Read-only
Applies to	`dagroup`
Data type	double
Values	Any integer in the range `[0 Inf]`. The default value is `0`.

See Also

Functions

`getdata`, `start`

Properties

`RecordsAcquired`, `RecordsToAcquire`

RecordsToAcquire

Number of records to acquire for logging session

Description

`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`.

Characteristics

Access	Read-only while logging
Applies to	`dagroup`
Data type	double
Values	Any integer in the range `[0 Inf]`. The default value is `120`.

See Also

Properties

`Logging`, `RecordsAvailable`

ScanRate

Fastest possible data update rate

Description

`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 may not be attainable by the server due to network load, server load and other factors.

Characteristics

Access	Read-only while logging
Applies to	`dagroup`
Data type	double
Values	The value is set by the server when a `daitem` object is created or when you connect to the server.

See Also

Properties

`UpdateRate`

ServerID

Server identity

Description

`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.

You use `opcserverinfo` to find a list of available servers and their Server IDs.

Characteristics

Access	Read-only while connected
Applies to	`opcda`
Data type	`string`
Values	The default value is specified during object creation.

See Also

Functions

`opcda`, `opcserverinfo`

Properties

`Host`

ShutDownFcn

Callback function file to execute when OPC server shuts down

Description

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.

Fleld 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.

Characteristics

Access	Read/write
Applies to	`opcda`
Data type	string, function handle, or cell array
Values	The default value is `@opccallback`.

See Also

Functions

`opccallback`

Properties

EventLog

StartFcn

Callback function file to execute immediately before logging starts

Description

You configure `StartFcn` to execute a callback function file 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.

Fleld 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.

Characteristics

Access	Read/write
Applies to	`dagroup`
Data type	string, function handle, or cell array
Values	The default value is an empty matrix (`[]`).

See Also

Functions

`start`

Properties

`EventLog`, `Logging`

Status

Status of connection to OPC server

Description

`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 will be set to `disconnected`.

Characteristics

Access	Read-only
Applies to	`opcda`
Data type	`string`
Values	`[ { disconnected } | connected ]`

See Also

Functions

`connect`, `disconnect`

Properties

`ShutDownFcn`

StopFcn

Callback function file to execute immediately after logging stops

Description

You configure `StopFcn` to execute a callback function file 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.

Fleld 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.

Characteristics

Access	Read/write
Applies to	`dagroup`
Data type	string, function handle, or cell array
Values	The default value is an empty matrix (`[]`).

See Also

Functions

`stop`

Properties

`EventLog`, `RecordsAcquired`, `RecordsToAcquire`

Subscription

Enable server update when data changes

Description

`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 and/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.

Characteristics

Access	Read/write
Applies to	`dagroup`
Data type	string
Values	`[off	{ on }]`

See Also

Functions

`read`, `readasync`, `refresh`

Properties

`Active`, `DataChangeFcn`, `DeadbandPercent`, `UpdateRate`

Tag

Label to associate with OPC Toolbox object

Description

You configure `Tag` to be a string value that uniquely identifies an OPC Toolbox 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.

Characteristics

Access	Read/write
Applies to	`dagroup`, `daitem`, `opcda`
Data type	string
Values	The default value is an empty string ( ).

See Also

Functions

`opcfind`

TimeBias

Time bias of group

Description

`TimeBias` indicates the time difference between the server and client machines. In some cases the data may have been collected by a device operating in a time zone other than that of the client. Then it will be useful to know what the time of the device was at the time the data was collected (e.g., to determine what shift was on duty at the time).

The time is specified in minutes and can be positive or negative.

Characteristics

Access	Read-only
Applies to	`dagroup`
Data type	double
Values	The default value is `0`.

See Also

Properties

`TimeStamp`

Timeout

Maximum time to wait for completion of instruction to server

Description

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 time-out occurs, the read or write operation aborts. 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.

Characteristics

Access	Read/write
Applies to	`opcda`
Data type	double
Values	Any value in the range `[0 Inf]`. The default value is `10`.

See Also

Functions

`read`, `readasync`, `write`, `writeasync`

TimerFcn

Callback function file to execute when predefined period passes

Description

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.

Characteristics

Access	Read/write
Applies to	`opcda`
Data type	string, function handle, or cell array
Values	The default value is an empty matrix ([ ]).

See Also

Functions

`connect`, `disconnect`

Properties

`TimerPeriod`

TimerPeriod

Period between timer events

Description

`TimerPeriod` specifies the time, in seconds, that must pass before the callback function specified by `TimerFcn` is called.

Some timer events may not be processed if your system is significantly slowed or if the `TimerPeriod` value is too small.

Characteristics

Access	Read only while logging
Applies to	`opcda`
Data type	double
Values	Any value in the range `[0.001 Inf]`. The default value is `10`.

See Also

Functions

`connect`, `disconnect`

Properties

`TimerFcn`

TimeStamp

Time when item was last read

Description

`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 convert date vectors to date strings with the `datestr` function, and to MATLAB date numbers with the `datenum` function.

Characteristics

Access	Read-only
Applies to	`daitem`
Data type	MATLAB date vector
Values	The default value is an empty matrix (`[]`).

See Also

Functions

`datestr`, `datenum`, `datevec`, `read`, `readasync`, `refresh`

Properties

`Quality`, `Subscription`, `UpdateRate`, `Value`

Type

OPC Toolbox object type

Description

`Type` indicates the type of the object. The OPC Toolbox 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 Toolbox objects of a given type using the `opcfind` function and the `Type` value.

Characteristics

Access	Read-only
Applies to	`dagroup`, `daitem`, `opcda`
Data type	string
Values	The value is set during object creation.

See Also

Functions

`opcfind`

UpdateRate

Rate, in seconds, at which subscription callbacks occur

Description

`UpdateRate` specifies the rate, in seconds, at which subscription callbacks occur. Therefore, `UpdateRate` 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.

Note that 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.

Characteristics

Access	Read-only while logging
Applies to	`dagroup`
Data type	double
Values	Any value greater than or equal to `0`. The default value is `0.5`.

See Also

Functions

`start`

Properties

`Active`, `DeadbandPercent`, `Subscription`

UserData

Data to associate with OPC Toolbox object

Description

You configure `UserData` to store data that you want to associate with an OPC Toolbox object. The object does not use this data directly, but you can access it using the `get` function.

Characteristics

Access	Read/write
Applies to	`dagroup`, `daitem`, `opcda`
Data type	Any MATLAB data type
Values	The default value is an empty matrix (`[ ]`).

See Also

Properties

`Tag`

Value

Item value

Description

`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.

Characteristics

Access	Read-only
Applies to	`daitem`
Data type	Any MATLAB data type
Values	The default value is an empty matrix (`[ ]`).

See Also

Functions

`read`, `readasync`, `refresh`

Properties

`Active`, `DataType`, `DeadbandPercent`, `Quality`, `Subscription`, `TimeStamp`, `UpdateRate`

WriteAsyncFcn

Callback function file to execute when asynchronous write completes

Description

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.

Fleld 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.

Fleld Name	Description
`ItemID`	The item name.

Write async event information is stored in the `EventLog` property.

Characteristics

Access	Read/write
Applies to	`dagroup`

Data type	string, function handle, or cell array
Values	The default value is `@opccallback`.

See Also

Functions

`opccallback`, `writeasync`

Properties

`EventLog`

Historical Data Access User s Guide

12

Introduction to OPC Historical Data Access (HDA)

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. OPC Toolbox software provides a client interface to historical data access servers via the MATLAB environment. This client interface lets you:

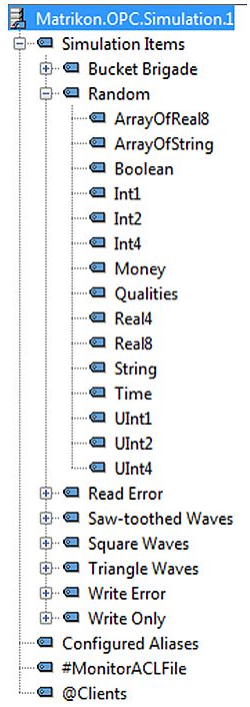
- Retrieve data from HDA servers into MATLAB
- Preprocess that data for common analysis tasks
- Visualize the data for easy interpretation

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.

OPC Toolbox provides capabilities for reading raw and processed data from 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

To interact with an OPC server, OPC Toolbox software needs:

- 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`	Text string 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.

OPC HDA Objects

OPC 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.

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 the Matrikon OPC Simulation Server" on page 1-19. The code requires only minor changes to work with other servers.

Create an HDA Client Object

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" on page 1-21. 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 string.
`FullyQualifiedID`	The fully qualified item ID of the node, as a string. 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)

                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)

                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 Toolbox software.

Using OPC Toolbox HDA Client Objects

OPC Toolbox HDA Objects

OPC 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 Toolbox 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 text string, 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, a client's 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
Methods
```

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
```

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.

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 string.
`FullyQualifiedID`	The fully qualified item ID of the node, as a string, 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)

                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 using OPC Toolbox software.

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, refer to Appendix C.

Reading OPC Historical Data

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 ));
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
```

15

Working with OPC HDA Data Objects

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 8-16.

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, OPC 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 Strings” 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

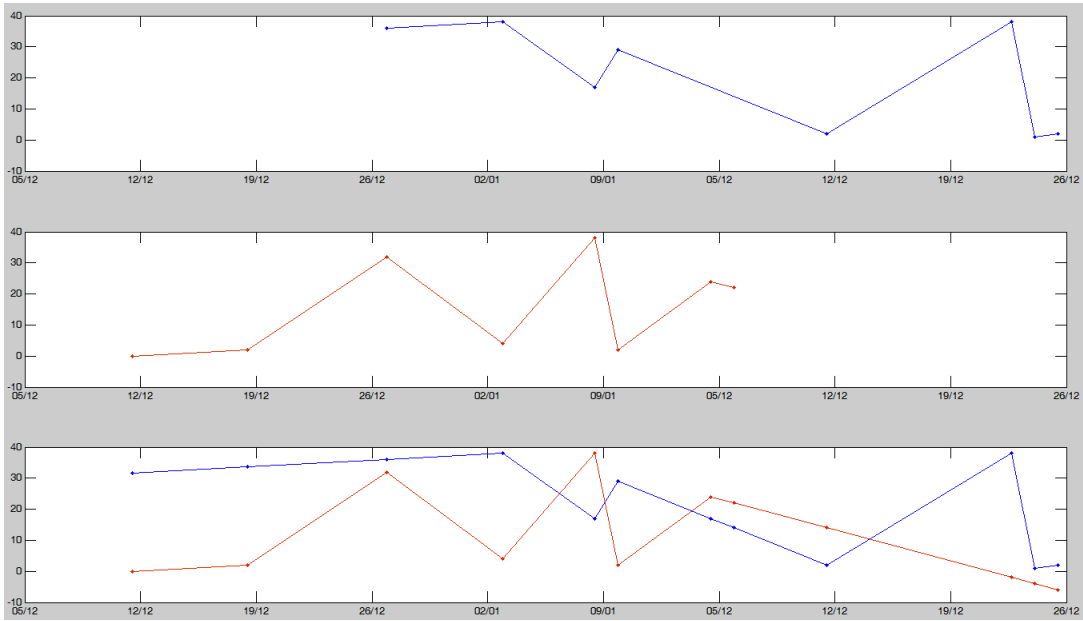
Manipulate Data Using OPC Toolbox HDA Objects

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

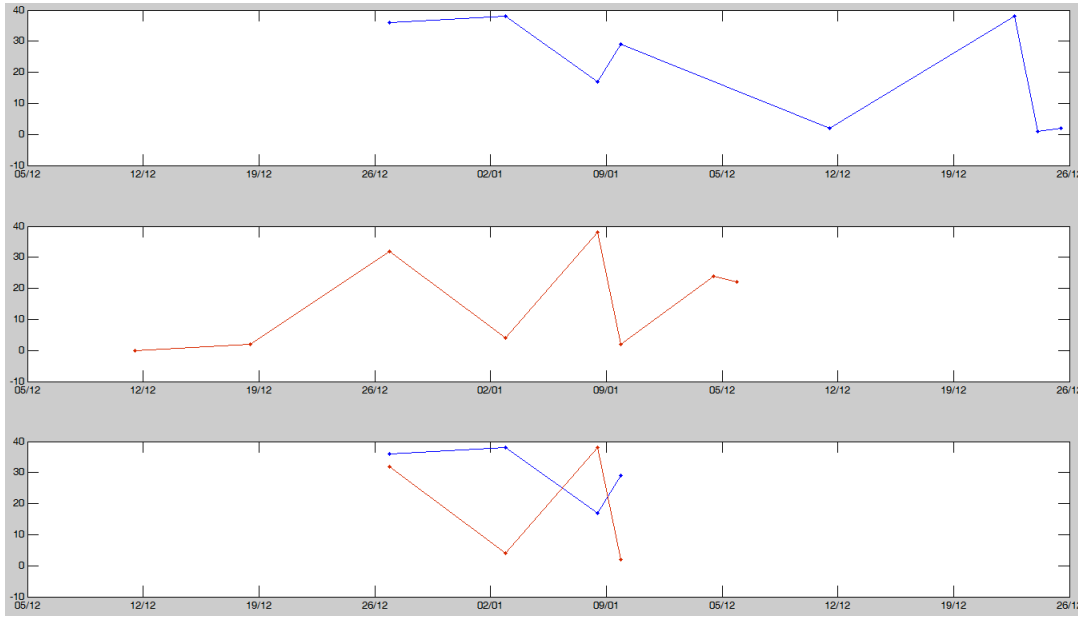
Given an array of data objects, `tsunion` adapts all data to have a single common set of timestamps by finding all unique time stamps in all items of the array. The values of each data item are then extrapolated or interpolated at the new timestamps. Resampling is performed using the method specified in the function call. Valid methods are `linear`, `spline`, `pchip`, `nearest`, and `hold`. The default is `linear`. If any returned `Value` is a string, 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.



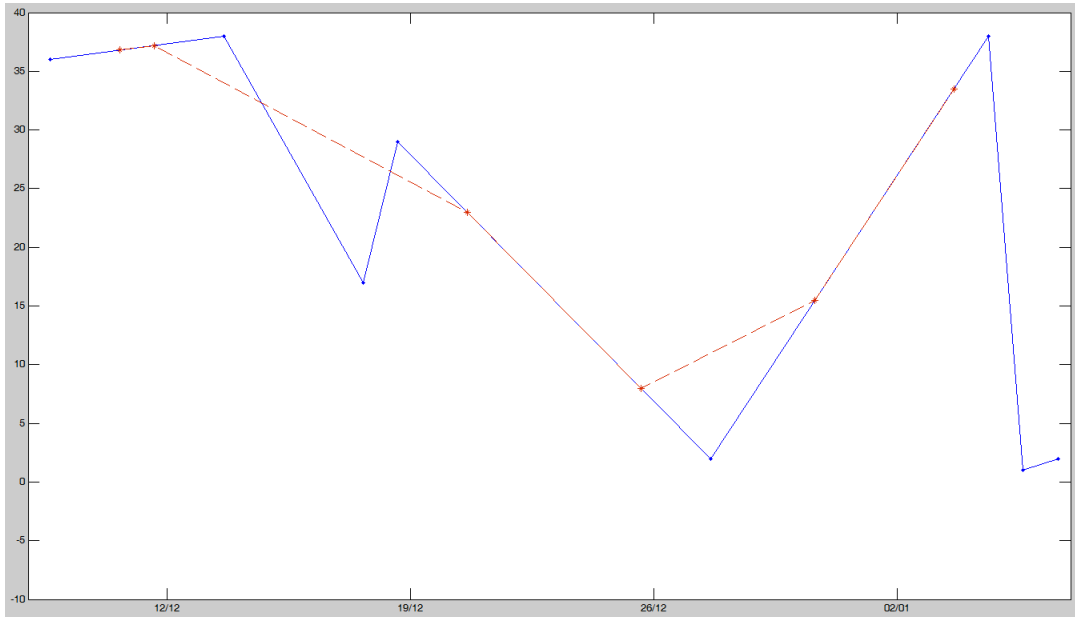
The previous figure shows how the values of two data objects, plotted in the first and second positions respectively, can be intersected to produce a new object whose elements contain only timestamps common to the original two. Uncommon timestamps are discarded along with their data values.

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 string values, only the `hold` method is supported. Trying to resample data containing strings 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 Toolbox 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`.

16

OPC HDA and UA Classes — Alphabetical List

opc.hda.AggregateTypes

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

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

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

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`

opc.ua.AggregateFnId class

OPC UA node aggregate function ID enumeration

Description

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 string 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.
`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.

Function	Description
`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.
`StandardDeviationPopul`	Retrieve the standard deviation for the interval for a complete population (n) which includes Simple Bounding Values.
`StandardDeviationSampl`	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.
`StartBound`	Retrieve the value at the beginning of the interval using Simple Bounding Values.

Function	Description
`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.

See Also

Functions

`opcua` | `opcuaserverinfo` | `readProcessed`

Classes

opc.ua.Client | opc.ua.Node | opc.ua.ServerInfo

opc.ua.Client class

OPC UA client object

Description

Use the OPC UA Client to connect to an OPC UA server, browse the server's name space, and read and write data from and to the server.

Construction

Use the `opcua` function to construct clients in your MATLAB session associated with a particular OPC UA server.

Methods

Properties

Property	Description
`Hostname`	Server host name or IP address
`Port`	Port number used for TCP/IP connections to the server
`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

Property	Description
`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

See Also

Functions

`opcua` | `opcuaserverinfo`

Classes

opc.ua.AggregateFnId | opc.ua.Node | opc.ua.ServerInfo

opc.ua.Node class

OPC UA node object

Description

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.

Construction

Use the `opcuanode` function to construct nodes in your MATLAB session.

Methods

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 string or integer, depending on the `IdentifierType`.

Property	Description
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`	String that uniquely describes this node.
Essential attributes	
`Description`	String 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`.
`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.

See Also

Functions

`opcua` | `opcuanode` | `opcuaserverinfo`

Classes

opc.ua.Client | opc.ua.ServerInfo

More About

- “OPC UA Server Data Types” on page 17-6

opc.ua.ServerInfo class

OPC UA server information object

Description

The `opc.ua.ServerInfo` class stores information about installed OPC UA servers on a specified host. You can use the `opcua` function to quickly construct clients associated with a particular OPC UA server.

Construction

You should not directly create objects in this class. Instead, use `opcuaserverinfo` to retrieve information about servers from a particular host.

Methods

Properties

Property	Description
`Hostname`	Host name used by the server to authenticate connections
`Port`	Port number used for connections to the server
`Description`	Friendly name of the OPC UA server

See Also

Functions

`opcua` | `opcuaserverinfo`

Classes
opc.ua.Client | opc.ua.Node

Unified Architecture User's Guide

OPC Unified Architecture (UA)

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 OPC 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. OPC 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 OPC UA Server reference for information on how those functions work. OPC Toolbox provides a client interface to the OPC UA servers which allows you to browse the server namespace to find nodes of interest. For some of the tasks you can perform with OPC Toolbox, see the related examples at the end of this topic.

Related Examples

- Read and Write Current OPC UA Server Data
- Read Historical OPC UA Server Data
- Visualize and Preprocess OPC UA Data

OPC UA Components

In this section...

Overview

OPC 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 connect the client to the server using `connect`. The Client includes a number of properties describing the server capabilities. See opc.ua.Client for more information on the properties exposed by 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 may 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's `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 may 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.

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 (dependant 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 OPC 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 string	Arrays converted to cell array of strings
Int16	int16	
Int32	int32	
Int64	int64	

OPC UA Data Type	MATLAB Data Type	Notes
LocalizedText	String	Arrays converted to cell array of strings
NodeId (*)	Encoded string	Arrays converted to cell array of strings
QualifiedName (*)	Encoded string	Arrays converted to cell array of strings
SByte	int8	
String	String	Arrays converted to cell array of strings
Structure (*)	Structure	
Time (*)	Datetime	Arrays not supported.
UInt16	uint16	
UInt32	uint32	
UInt64	uint64	
XmlElement (*)	String	Arrays converted to cell array of strings

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 (*).

Access Data from OPC UA Servers

In this section...

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 the OPC Foundation Quickstart Historical Access Server.

Note To run the sample code in the following steps you need the OPC Foundation Quickstart Historical Access Server running on your local machine. For installation details, see Install an OPC UA Simulation Server for OPC UA Examples. 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 Toolbox 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.

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 OPC Foundation Quickstart Historical Access Server is `opc.tcp://localhost:62550/Quickstarts/HistoricalAccessServer`.

OPC UA Server URLs are advertised through an OPC UA Discovery Service, 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.

Use the `opcuaserverinfo` function to query hosts from the command line.

```
serverList = opcuaserverinfo( localhost )
```

```
serverList =
1x2 OPC UA ServerInfo array:
    index              Description              Hostname  Port
    -----  -----------------------------------  --------  -----
      1    Quickstart Historical Access Server  ons-dean  62550
      2    UA Sample Server                     ons-dean  51210
```

Locate the server of interest by using the `findDescription` function to search for a specific string. In this example you connect to the Historical Access Server.

```
hsInfo = findDescription(serverList,  Historical )
```

```
hsInfo =
OPC UA ServerInfo  Quickstart Historical Access Server :

   Connection Information
    Hostname:  ons-dean
        Port: 62550
```

From this discovery process, you can identify the port (62550) on which the OPC UA server listens for connections.

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 OPC UA ServerInfo result to construct an OPC UA client directly. (You could also create a client using the hostname and port provided by the OPC UA ServerInfo properties.)

```
uaClient = opcua(hsInfo)
```

```
uaClient =
OPC UA Client Quickstart Historical Access Server:
                      Hostname: ons-dean
                          Port: 62550
                       Timeout: 10
                        Status: Disconnected
```

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
uaClient =
OPC UA Client Quickstart Historical Access Server:
                      Hostname: ons-dean
                          Port: 62550
                       Timeout: 10
                        Status: Connected
                   ServerState: Running
                 MinSampleRate: 0 secs
           MaxHistoryReadNodes: 0
      MaxHistoryValuesPerNode: 0
                  MaxReadNodes: 0
                 MaxWriteNodes: 0
```

The additional properties describe capabilities of the server, notably limits for various read and write operations. A limit value of 0 means the server does not impose a direct limit on that capability.

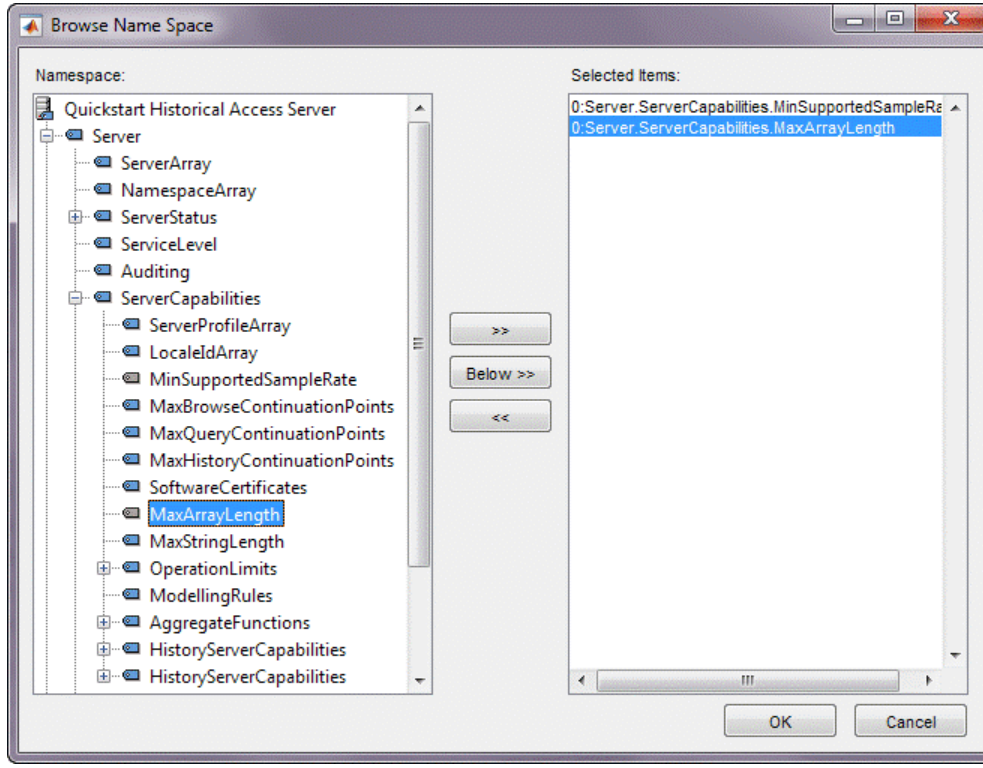
Step 3: Browse OPC UA Server Namespace

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, string, or Globally Unique Identifer 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.

OPC 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 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"

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 OPC 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 =
    [0 secs]
    [     0]
ts =
   30-Jun-2015 05:05:13
   30-Jun-2015 05:05:13
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".

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 Sample parent node.

```
sampleNode = findNodeByName(uaClient.Namespace,  Sample )

sampleNode =
OPC UA Node object:
                      Name: Sample
               Description:
            NamespaceIndex: 2
                Identifier: Sample
                  NodeType: Object
                    Parent: Archive
                  Children: 14 nodes.
```

The Sample node is an Object node, so it has no Value. However, it has 14 Children. Locate the Double and Int32 child nodes.

```
doubleNode = findNodeByName(sampleNode,  Double );
int32Node = findNodeByName(sampleNode,  Int32 )

int32Node =
OPC UA Node object:
                      Name: Int32
               Description:
            NamespaceIndex: 2
                Identifier: 1:Quickstarts.HistoricalAccessServer.Data.Sample.Int32.txt
                  NodeType: Variable
                    Parent: Sample
                  Children: 2 nodes.
            ServerDataType: Int32
```

```
    AccessLevelCurrent: read
    AccessLevelHistory: read/write
           Historizing: 0
```

Although the Int32 (and Double) 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.

```
dataToday = readHistory(uaClient, [doubleNode, int32Node], datetime( today ), datetime( now ))
```

```
dataToday =
1-by-2 OPC UA Data object:
     Name        Value           Start Timestamp           End Timestamp            Quality
    ------  ---------------  -----------------------  -----------------------  ------------------
    Double  9 double values  2015-06-30 04:00:10.000  2015-06-30 04:01:30.000  3 unique qualities
    Int32   12 int32 values  2015-06-30 04:00:02.000  2015-06-30 04:01:30.000  3 unique qualities
```

Step 6: Plot the Data

You can plot the data directly from the resulting OPC UA Data object.

```
plot(dataToday)
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".

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. Note that if you clear an OPC UA Client from memory, the connection to the server is automatically closed.

OPC Information Reference

A

OPC Quality Strings

OPC Quality Strings

OPC Toolbox software uses specific quality values 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 strings 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 strings are layered on top of OPC DA quality strings.

The following topics describe the OPC quality values and strings associated with each quality part.

For more information on OPC quality strings, 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

OPC Toolbox software uses the following major quality values and strings. The major quality is contained in bits 7 and 8 of the quality value.

Major Quality Strings Used in OPC Toolbox Software

Value	Quality String	Description
0	`Bad`	The value is not useful for the reason indicated by the substatus. The table Bad Quality Substatus Strings 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 Strings contains information about the substatus for uncertain quality.
3	`Good`	The quality of the value is good. The table Good Quality Substatus Strings 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.

More About

- “OPC Quality Strings” on page A-2
- “Quality Substatus” on page A-4
- “Limit Status” on page A-7

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 Strings
- Uncertain Quality Substatus Strings
- Bad Quality Substatus Strings

Good Quality Substatus Strings

Value	Substatus String	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 Strings

Value	Substatus String	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` string 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.

Value	Substatus String	Description
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 Strings

Value	Substatus String	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.

Value	Substatus String	Description
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.

More About

- "OPC Quality Strings" on page A-2
- "Major Quality" on page A-3
- "Limit Status" on page A-7

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 strings used in OPC Toolbox software.

Value	Limit Status String	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 string 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.

More About

- “OPC Quality Strings” on page A-2
- “Major Quality” on page A-3
- “Quality Substatus” on page A-4

B

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:

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.

More About

- "OPC DA Server Item Properties" on page B-2
- "OPC Specific Properties" on page B-5
- "OPC Recommended Properties" on page B-7

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

More About

- "OPC DA Server Item Properties" on page B-2
- "OPC Item Property Set" on page B-3

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 a string 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 a string to be associated with this contact when it is in the open (zero) state.
108	“Item Timezone”

PropID	Description
	The difference in minutes between the item's UTC Timestamp and the local time in which the item value was obtained. OPC 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 string providing a brief set of instructions for the operator to follow when this alarm occurs.

PropID	Description
302	“Alarm Area List” An array of strings indicating the plant or alarm areas which include this item.
303	“Primary Alarm Area” A string indicating the primary plant or alarm area including this item.
304	“Condition Logic” An arbitrary string 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.

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-5

C

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 a free-form string.
- **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.

Functions — Alphabetical List

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.

GrpObj is a dagroup object. By default, GrpObj has the Active property set to `on`, GroupType set to `private`, and the Subscription property set to `on`.

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 with 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 Obj.

`GrpObj = addgroup(DAObj, GName , GrpType )` adds a group to the opcda object DAObj with the group type specified by *`GrpType`*. 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` then 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

running `opcserverinfo(DAObj)` and checking the `SupportedInterfaces` field for the `IOPCServerPublicGroups` interface.

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 `opcda` client:

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

Create a group using a default group name:

```
grp1 = addgroup(da);
```

Add another group, providing the name:

```
grp2 = addgroup(da,  AddgroupEx );
```

See Also

`additem` | `opcserverinfo`

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. 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 strings, one for each item ID. *`DataType`* is the data type in which the item's value will be 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 strings, one for each item ID. *`Active`* can be `on` or `off` . The active status is stored in the `Active` property.

Examples

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 )
```

See Also

`getnamespace` | `serveritems`

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 opcdemoHDAData;
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 (opcda)

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

array of OPC DA items

Initial list of OPC DA items, specified as an array of OPC DA items.

`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 items

array of OPC DA items

List of OPC DA items, returned as an array of OPC DA items.

See Also

`addgroup` | `additem` | `getnamespace`

Introduced in R2013a

browseNameSpace (opchda)

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

array of OPC HDA items

Initial list of OPC HDA items, specified as an array of OPC HDA items.

`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 items

array of OPC HDA items

List of OPC HDA items, returned as an array of OPC HDA items.

See Also

`getNameSpace (opchda)` | `readattime` | `readmodified` | `readprocessed` | `readraw`

Introduced in R2013a

browseNamespace (opcua)

Graphically browse name space and select nodes from 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
```

See Also

`getNamespace` | `readAtTime` | `readHistory` | `readProcessed` | `readValue` | `writeValue`

Introduced in R2015b

cancelasync

Cancel asynchronous read and write operations

Syntax

```
cancelasync(GObj)
cancelasync(GObj,TransID)
```

Description

`cancelasync(GObj)` cancels all asynchronous read or write operations that are in progress for the group object specified by `GObj`. Note that 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 operation(s), specified by the transaction ID(s) given by `TransID`. You can cancel specific asynchronous requests using this syntax.

Examples

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)
```

See Also

`readasync` | `writeasync`

clearеventlog

Clear event log, discarding all events

Syntax

```
clearеventlog(DAObj)
```

Description

`clearеventlog(DAObj)` clears the event log for `opcda` object `DAObj`. `DAObj` can be an array of objects. `clearеventlog` 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:

```
clearеventlog(da)
el2 = da.EventLog
```

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 );
```

See Also

copyobj | makepublic

connect

Connect OPC Toolbox client to server

Syntax

```
connect(Obj)
```

Description

`connect(Obj)` connects the `opcda` or `opchda` object `Obj` to the OPC server that you specified by the `Host` and `ServerID` properties. When you connect `Obj`, the `Status` property takes the value `connected`. You can disconnect `Obj` from the server with the `disconnect` function. When you disconnect `Obj`, the `Status` property takes the value `disconnected`.

If `Obj` is an array of objects and the function cannot connect some of these objects, it generates a warning message. If the function can connect none of the objects, it generates an error message.

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

Create a Data Access client and connect to the server:

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

Create an HDA client for the Matrikon Simulation Server and connect to the server:

```
hdaObj = opchda( localhost , Matrikon.OPC.Simulation );
connect(hdaObj);
```

See Also

`disconnect` | `isConnected`

connect (opcua)

Connect OPC UA client to server

Syntax

```
connect(UaClient)
```

Description

`connect(UaClient)` connects the OPC UA client `UaClient` to its referenced server using anonymous user authentication.

Note OPC Toolbox currently supports only unsecured connections.

When the client successfully connects to the server, the client `Status` property is set to `Connected`, the first level of the server's namespace is retrieved, and various essential properties of the client are read from the server.

If `UaClient` is a vector of clients and some clients can connect but some cannot, a warning is issued. If no clients can connect, an error is generated.

Use the `disconnect` function to disconnect the client from the server.

Examples

Locate a server and connect a client to it.

```
s = opcuaserverinfo( localhost );
UaClient = opcua(s);
connect(UaClient);
```

See Also

`isConnected` | `disconnnect` | `opcua`

Introduced in R2015b

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 Toolbox 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's name, and note that the second group's name is unchanged:

```
grp1.Name =  NewGroupName ;
grp2.Name
```

See Also

`obj2mfile` | `propinfo`

delete

Remove OPC Toolbox objects from memory

Syntax

```
delete(Obj)
```

Description

`delete(Obj)` removes the OPC Toolbox 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
```

See Also

clear | disconnect | isvalid | opc.hda.reset

disconnect

Disconnect OPC Toolbox client from server

Syntax

```
disconnect(Obj)
```

Description

`disconnect(Obj)` disconnects the OPC Toolbox 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

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
```

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
```

And disconnect from the server:

```
disconnect(hdaObj);
hdaObj.Status
```

See Also

`connect` | `isConnected` | `propinfo`

disconnect (opcua)

Disconnect OPC UA client from its 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
```

See Also

`connect` | `isConnected` | `opcua`

Introduced in R2015b

disp

Summary of information for OPC Toolbox objects

Syntax

```
Obj
disp(Obj)
```

Description

`Obj` or `disp(Obj)` displays summary information for OPC Toolbox 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 string 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 opcdemoHDAData;
disp(hdaDataSmall)
```

See Also

`addgroup` | `additem` | `opcda` | `showValues`

double

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to double type matrix

Syntax

```
V = double(DObj)
```

Description

`V = double(DObj)` converts the OPC HDA data object array `DObj` into a matrix of type double. `V` is constructed as an `M`-by-`N` array of doubles, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`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

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create a double matrix from the result:

```
load opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vDouble = double(dUnion);
```

See Also

`resample` | `tsintersect` | `tsunion`

findDescription

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

`opchdaserverinfo`

findDescription (opcua)

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 string `DescStr`.

Examples

This example shows how to 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
```

See Also

`opcuaserverinfo`

Introduced in R2015b

findNodeById (opcua)

Find 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 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 object:
                     Name: ServerCapabilities
              Description: Describes capabilities supported by the server.
           NamespaceIndex: 0
               Identifier: 2268
                 NodeType: Object

                   Parent: Server
                 Children: 14 nodes.
```

See Also

`findNodeByName` | `opcua`

Introduced in R2015b

findNodeByName (opcua)

Find 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 object:
                      Name: ServerCapabilities
               Description: Describes capabilities supported by the server.
           NamespaceIndex: 0
               Identifier: 2268
                 NodeType: Object

                   Parent: Server
                 Children: 14 nodes.
```

See Also

`findNodeById` | `opcua`

Introduced in R2015b

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)
```

See Also

`getnamespace` | `serveritems`

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 OPC 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
```

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)
```

See Also

`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)
```

See Also

`genslread`

get

OPC Toolbox 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 `PropName` for the OPC Toolbox object `Obj`.

If `PropName` is a cell array of strings containing 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)
```

More About

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;
```

See Also

`opchelp` | `propinfo` | `set`

getAllChildren (opcua)

Recursively retrieve all children of 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
```

See Also

`findNodeByName` | `getNamespace` | `findNodeById` | `browseNamespace`

Introduced in R2015b

getDescription

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 string associated with the aggregate type or item attribute given by `ID`. If `ID` is a vector, `DStr` is a cell array of description strings.

`DStr = getDescription(Obj, NameStr)` returns the description string associated with the aggregate type or item attribute given by the string `NameStr`. If `NameStr` is a cell array of strings, `DStr` is a cell array of description strings.

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

`getIDFromName`

getdata

Retrieve logged records from OPC 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 OPC 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 string.
`Value`	The data value. The data type is defined by the item's `DataType` property.
`Quality`	The data quality, as a string. For a description, see “OPC Quality Strings” 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 strings 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 strings or arrays of values.

`Qual` is an `NRec`-by-`NItem` array of quality strings 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.

Argument	Behavior for Missing Items
`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
```

See Also

`flushdata` | `peekdata` | `start` | `stop`

getIDFromname

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 a cell array of strings, `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

`getDescription` | `getNameList`

getIDList

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

`getNameList`

getIndexFromID

Class: opc.hda.Data
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 strings.

Examples

Load the OPC HDA example data file and find the index of `Item Example.Item.2` :

```
load opcdemoHDAData;
ind = getIndexFromID(hdaDataVis,  Example.Item.2 );
```

getNameList

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 strings (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 (opcda)

OPC data access 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 following table:

BrowseFilter	Description
`StartItemID`	Specify the `FullyQualifiedID` of a branch node, as a 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.

BrowseFilter	Description
`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 string ( ) 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.

Examples

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)
```

Get only the first level of the name space:

```
nsPart = getnamespace(da, Depth ,1)
```

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);
```

See Also

additem | flatnamespace | serveritems

getNameSpace (opchda)

Package: opc.hda

OPC historical data access 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` level(s) 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 the entire name space for the Matrikon Simulation Server on the local host:

```
hdaObj = opchda( localhost , Matrikon.OPC.Simulation );
connect(hdaObj);
nsFull = getNameSpace(hdaObj)
```

Get only the first level of the name space:

```
nsPart = getNameSpace(hdaObj, Depth ,1)
```

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);
```

See Also

`connect`

getNamespace (opcua)

Namespace of 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.

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.

See Also

`browseNamespace`

Introduced in R2015b

getNodeAttributes (opcua)

Read 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 or cell array of strings

Server attributes specified as an array of uint32 or cell 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.

See Also
`browseNamespace` | `getNamespace` | `readValue`

Introduced in R2015b

getServerStatus (opcua)

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 string)
`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 string

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:
```

See Also

connect | opcuaserverinfo | disconnect | opcua

Introduced in R2015b

int16

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to int16 matrix

Syntax

```
V = int16(DObj)
```

Description

`V = int16(DObj)` converts the OPC HDA aata object array `DObj` into an int16 matrix. `V` is constructed as an `M`-by-`N` array of int16 values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`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

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 opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vInt16 = int16(dUnion);
```

See Also

`resample` | `tsintersect` | `tsunion`

int32

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to int32 matrix

Syntax

```
V = int32(DObj)
```

Description

`V = int32(DObj)` converts the OPC HDA data object array `DObj` into an int32 matrix. `V` is constructed as an `M`-by-`N` array of int32 values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`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

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 opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vInt32 = int32(dUnion);
```

See Also

`resample` | `tsintersect` | `tsunion`

int64

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to int64 matrix

Syntax

```
V = int64(DObj)
```

Description

`V = int64(DObj)` converts the OPC HDA data object array `DObj` into an int64 matrix. `V` is constructed as an `M`-by-`N` array of int64 values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`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

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 opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vInt64 = int64(dUnion);
```

See Also

`resample` | `tsintersect` | `tsunion`

int8

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to int8 matrix

Syntax

```
V = int8(DObj)
```

Description

`V = int8(DObj)` converts the OPC HDA Data object array `DObj` into an int8 matrix. `V` is constructed as an `M`-by-`N` array of int8 values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`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

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 opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vInt8 = int8(dUnion);
```

See Also

`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)
```

See Also

`connect` | `disconnect`

isConnected (opcua)

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
```

See Also

`connect` | `disconnect`

Introduced in R2015b

isObjectType (opcua)

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
```

See Also

`opcuanode` | `isVariableType`

Introduced in R2015b

isvalid

True for undeleted OPC Toolbox 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 Toolbox 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
```

See Also

`delete` | `opchelp`

isVariableType (opcua)

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
```

```
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
```

See Also

`opcuanode` | `isObjectType`

Introduced in R2015b

load

Load OPC Toolbox 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 Toolbox 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;
```

See Also

`opchelp` | `propinfo` | `save`

logical

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to logical matrix

Syntax

```
V = logical(DObj)
```

Description

`V = logical(DObj)` converts the OPC HDA data object array `DObj` into an logical matrix. `V` is constructed as an `M`-by-`N` array of logical values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`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

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create a logical matrix from the result:

```
load opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vLogical = logical(dUnion);
```

See Also

`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);
```

See Also

clonegroup | opcserverinfo

obj2mfile

Convert OPC Toolbox 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 Toolbox 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 are 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;
```

See Also

opchelp | propinfo

opccallback

Event information for OPC Toolbox callbacks

Syntax

```
opccallback(Obj,Event)
```

Description

`opccallback(Obj,Event)` displays a message in the MATLAB Command Window that contains information about an OPC Toolbox 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.

See Also

`showopcevents`

opcda

Construct OPC data access object

Syntax

```
Obj = opcda( Host , ServerID )
Obj = opcda( Host , ServerID , P1 ,V1, P2 ,V2,...)
```

Description

`Obj = opcda( Host , ServerID )` constructs an OPC data access (`opcda`) object, `Obj`, for the host specified by `Host` and the OPC server ID specified by `ServerID`. When you construct `Obj`, its initial `Status` property value is `disconnected` . To communicate with the server, you must connect `Obj` to the server with the `connect` function.

`Obj = opcda( Host , ServerID , P1 ,V1, P2 ,V2,...)` constructs an OPC data access object, `Obj`, 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.

Note that the property name/property value pairs can be any format that the `set` function supports, i.e., parameter-value string pairs, structures, and parameter-value cell array pairs.

At any time, you can view a complete listing of OPC Toolbox functions and properties with the `opchelp` function.

Examples

Create an `opcda` client for a local server:

```
daObj1 = opcda( localhost ,  Dummy.Server.ID );
```

Create an `opcda` client for a remote server:

```
daObj2 = opcda( ServerHost1 ,  OPCServer.ID );
```

See Also

connect | opchelp | set

opcDataAccessExplorer

Open OPC Data Access Explorer app

Syntax

```
opcDataAccessExplorer
opcDataAccessExplorer(SessionName)
```

Description

`opcDataAccessExplorer` opens the OPC Data Access Explorer app. The OPC Data Access Explorer app allows you to graphically browse the contents of an OPC server, view server item properties, and create and configure OPC Toolbox clients, groups and items. With the OPC Data Access Explorer app you can also read and write OPC data, configure and start a logging session, and export logged data to the workspace.

If you use the OPC Data Access Explorer app to configure clients, groups, and items, you can export these to the workspace, a MAT-file, or an OPC session file that you can import into the app later.

`opcDataAccessExplorer(SessionName)` opens OPC Data Access Explorer app and loads a previously saved OPC session file identified by `SessionName`. If you do not specify a file extension as part of `SessionName`, `.osf` is the default.

For an example of a session with this app, see “Access Data with OPC Data Access Explorer” on page 3-2.

See Also

`openosf`

opc.daQualityString

OPC data access part of quality ID as strings

Syntax

```
[MajorStr, SubStr, LimitStr] = opc.daQualityString(IDs)
```

Description

`[MajorStr, SubStr, LimitStr] = opc.daQualityString(IDs)` converts the data access (DA) portion of the OPC quality IDs in `IDs` to the major string `MajorStr`, substatus string `SubStr`, and limit string `LimitStr`.

If `IDs` is a vector, each of `MajorStr`, `SubStr`, and `LimitStr` are cell strings 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 opcdemoHDAData;
newObj = tsunion(hdaDataSmall);
[majorStr, subStr, limitStr] = opc.daQualityString(newObj.Quality);
```

See Also

`opc.hdaQualityString` | `opcqstr`

opc.daSupport

OPC Toolbox 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 string

Machine hosting OPC servers, specified as a string.

Data Types: `char`

`FileName` — File for output text
`opcsupport.txt` (default)

File for output text, specified as a string.

Data Types: `char`

`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
string

Path to file of results, returned as a string.

See Also

`opc.hdaSupport` | `opcda` | `opcserverinfo`

Introduced in R2013a

opcfind

Find OPC Toolbox 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 Toolbox 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 Toolbox 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 Toolbox objects:

```
allObjCell = opcfind;
```

Find all objects with the Tag `myopcTag` :

```
myOPC = opcfind( Tag , myopcTag )
```

Find all `daitem` objects:

```
itmCell = opcfind( Type , daitem )
```

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 disply format for OPC objects:

```
fmt = opc.getDateDisplayFormat
```

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

string

OPC HDA server host name specified as a string.

Example: `host-name`

Data Types: `char`

`ServerID` — Identifier of OPC HDA server

string

Identifier of OPC HDA server, specified as a string.

Example: `MyHDAServer.1`

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside single quotes ( ). You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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 string 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:

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` | `cell`

Output Arguments

`hdaObj` — OPC HDA client

OPC HDA client object

OPC HDA client, returned as an OPC HDA client object.

See Also

`opchdaserverinfo`

Introduced in R2013a

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., parameter-value string pairs, structures, and parameter-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));
```

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 string 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.

See Also

`opchda`

opc.hdaQualityString

OPC historical data access part of quality ID as strings

Syntax

```
QStr = opc.hdaQualityString(IDs)
```

Description

`QStr = opc.hdaQualityString(IDs)` converts the HDA portion of the OPC quality IDs in `IDs`.

If `IDs` is a vector, `QStr` is a cell array of strings, 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 opcdemoHDAData;
newObj = tsunion(hdaDataSmall);
hdaQStr = opc.hdaQualityString([newObj.Quality]);
```

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
```

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.

`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 string 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 string `Matrikon` .

```
mIndex = findDescription(sInfo, Matrikon )
```

Construct an OPC HDA client for that server.

```
hdaClient = opchda(sInfo(mIndex))
```

See Also

opchda

opc.hdaSupport

OPC Toolbox 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 string

Machine hosting OPC servers, specified as a string.

Data Types: `char`

`FileName` — File for output text
`opcsupport.txt` (default)

File for output text, specified as a string.

Data Types: `char`

`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
string

Path to file of results, returned as a string.

See Also

`opc.daSupport` | `opchda` | `opcserverinfo`

Introduced in R2013a

opchelp

Help for OPC Toolbox 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 Toolbox data access functions with a brief description of each function.

`opchelp( Name )` displays online help for the function or property, `Name`. If `Name` is an OPC Toolbox 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 an OPC Toolbox 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 string `Out`.

`opchelp(Obj)` displays a complete listing of functions and properties for the OPC Toolbox object `Obj`, along with the online help for the object's 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 string `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 Toolbox 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 Toolbox `set` function:

```
opchelp set
```

Display help on the `opcda` object's `disconnect` function:

```
opchelp opcda/disconnect
```

Create an `opcda` object and queries help information on that object. The object's `Timeout` and `Status` properties are also queried.

```
da = opcda( localhost , Matrikon.OPC.Simulation );
opchelp(da)
timeoutHelp = opchelp(da, Timeout );
opchelp(da, Status );
```

See Also

`propinfo`

opcqid

Construct quality ID from item's quality string

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 string. The quality string must be in the form *`Major Quality`*`:` *`Quality Sub-status`* `(`*`Limit Status`*`)` .

If `QualityStr` is a cell array of quality strings, `QualityID` will be a matrix having the same size as `QualityStr`.

For more information on quality values, see “OPC Quality Strings” on page A-2.

Examples

Construct the quality ID from the quality string 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)
```

See Also

`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 Strings” 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)
```

See Also

`get` | `opcqstr`

opcqstr

Convert OPC quality ID into readable string

Syntax

```
QualityStr = opcqstr(QualityID)
```

Description

`QualityStr = opcqstr(QualityID)` constructs a quality string from a quality ID, stored in the `QualityID` property of a `daitem` object. The string 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 Strings” on page A-2.

Examples

Construct the quality string 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)
```

See Also

`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 string.
`Value`	The data value. The data type is dependent on the original Item's `DataType` property.
`Quality`	The data quality, as a string.
`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.

Property Name	Property Value
`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 string or cell array of strings. 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 strings 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 strings or arrays of values.

`Q` is an `NRec`-by-`NItem` array of quality strings 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
```

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 OPC Toolbox software 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 wish 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 Toolbox objects that you have previously created in this MATLAB session before you can run `opcregister`. If you attempt to run `opcregister` and OPC Toolbox 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, http://www.opcfoundation.org.

See Also

`opcreset`

opcreset

Disconnect and delete all OPC Toolbox objects

Syntax

```
opcreset
opcreset -force
```

Description

`opcreset` disconnects and deletes all OPC Toolbox 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 Toolbox blocks are open. Before calling `opcreset`, close all OPC Data Access Explorer sessions and all open Simulink models containing OPC Toolbox blocks.

`opcreset -force` closes all OPC Data Access Explorer sessions and all Simulink models containing OPC Toolbox 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 Toolbox 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
```

See Also

clear | delete | opcfind | opcDataAccessExplorer

opcserverinfo

Version, server, and status information

Syntax

```
Out = opcserverinfo
Out = opcserverinfo( Host )
Out = opcserverinfo(DAObj)
```

Description

`Out = opcserverinfo` returns a structure, `Out`, that contains information about OPC Toolbox and MATLAB software, including product version numbers.

`Out = opcserverinfo( Host )` returns a structure, `Out`, 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, `Out`, 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, as well as 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)
```

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 or string as defined by `datestr`. The date display format persists across MATLAB sessions.

`opc.setDateDisplayFormat( default )` resets the date display format to the string `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 opcdemoHDAData;
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 )
```

See Also

`opc.setDateDisplayFormat`

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 strings 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 by OPC Toolbox software 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 string *`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
```

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 timeseries 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 quality strings 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});
```

See Also

`getdata` | `opcread` | `opcstruct2array`

opcsupport

OPC Toolbox 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 string

Machine hosting OPC servers, specified as a character string.

Data Types: `char`

`FileName` — File for output text
`opcsupport.txt` (default)

File for output text, specified as a character string.

Data Types: `char`

`da` — Indicate data access only
literal string

Indicate data access only, specified as a literal string.

Data Types: `char`

`hda` — Indicate historical data access only
literal string

Indicate historical data access only, specified as a literal string.

Data Types: `char`

Output Arguments

`outFile` — Path to file of results

string

Path to file of results, returned as a string.

See Also

`opc.daSupport` | `opc.hdaSupport` | `opcserverinfo`

Introduced before R2006a

opcua (opcua)

Construct OPC UA client object

Syntax

```
UaClient = opcua(ServerInfoObj)
UaClient = opcua(ServerUrl)
UaClient = opcua(Hostname,Portnum)
```

Description

`UaClient = opcua(ServerInfoObj)` constructs an OPC UA client associated with the server referenced by `ServerInfoObj`. You can create server objects with the `opcuaserverinfo` function.

`UaClient = opcua(ServerUrl)` constructs a client associated with the server referenced by the URL string provided in `ServerUrl`. The server URL must use the `opc.tcp` protocol; OPC Toolbox does not support http or https connections.

`UaClient = opcua(Hostname,Portnum)` constructs an OPC UA client object associated with the server at port `Portnum` on the machine identified by `Hostname`. `Hostname` can be an IP address, or host name (short or fully qualified) specified as a character string. The client attempts to retrieve available endpoints, but does not error if the endpoints cannot be retrieved.

Note Some OPC UA servers require security for any connection to that server. OPC Toolbox supports only anonymous, unsecured connections to servers.

Examples

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 51210 on the local host.

```
UaClient = opcua( localhost ,51210)
```

```
UaClient =
OPC UA Client UA Sample Server:
    Hostname: localhost
        Port: 51210
     Timeout: 10

      Status: Disconnected
```

See Also

Functions

`connect` | `disconnect` | `opcuaserverinfo`

Classes

opc.ua.Client

Introduced in R2015b

opcuanode (opcua)

Construct OPC UA node objects

Syntax

```
NodeList = opcuanode(Index,Id)
NodeList = opcuanode(Index,Id,UaClient)
```

Description

`NodeList = opcuanode(Index,Id)` constructs 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 string, scalar integer, or cell array containing strings and scalar integers. Use this syntax to construct node objects for known nodes on an OPC UA server. The 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`.

You should construct node objects only when you know the index and identifier of nodes you are interested in. For nodes you need to find from the server, you can construct 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

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.

myNode =
OPC UA Node object:
                      Name: 2:10225
               Description:
            NamespaceIndex: 2
                Identifier: 10225
                  NodeType: Unknown

writeValue(UaClient,myNode,pi)
myNode3

myNode3 =
OPC UA Node object:
                      Name: 2:10225
               Description:
            NamespaceIndex: 2
                Identifier: 10225
                  NodeType: Variable

                  Children: 0 nodes.

            ServerDataType: Float
        AccessLevelCurrent: read/write
        AccessLevelHistory: none
               Historizing: 0
```

Construct 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
```

See Also

Functions

`browseNamespace` | `findNodeById` | `findNodeByName` | `getNamespace` | `opcua` | `readValue` | `writeValue`

Classes

opc.ua.Node

Introduced in R2015b

opcuaserverinfo (opcua)

Query host for installed OPC UA servers

Syntax

```
S = opcuaserverinfo( HostName )
```

Description

`S = opcuaserverinfo( HostName )` queries the host named `HostName` for its installed OPC UA servers. `HostName` can be a host name, or IP address, specified as a character string.

`S` is returned as an OPC UA ServerInfo object, or an array of these objects, containing the read-only properties `Description`, `Hostname`, and `Port`.

Use the `opcua` function to construct an OPC UA Client object directly from an OPC UA ServerInfo object.

Examples

This example shows how to find all available servers on the local host.

```
localServers = opcuaserverinfo( localhost );
localServers(1)
```

```
OPC UA ServerInfo  UA Sample Server :

   Connection Information
    Hostname:  HOST4421
        Port: 51210
```

See Also

Functions

`opcua`

Classes

opc.ua.ServerInfo

Introduced in R2015b

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.

See Also

`opcDataAccessExplorer` | `open`

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 OPC Toolbox software 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
```

See Also

`flushdata` | `getdata` | `start` | `stop`

plot

Class: opc.hda.Data
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 opcdemoHDAData;
plot(hdaDataVis)
```

See Also

`datetick` | `plot` | `stairs`

propinfo

Property information for OPC Toolbox objects

Syntax

```
Out = propinfo(Obj)
Out = propinfo(Obj, PropName )
```

Description

`Out = propinfo(Obj)` returns a structure array, `Out`, with field names given by the property names for `Obj`. Each property name in `Out` 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 string 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 strings, a cell array of structures is returned for each property.

Examples

```
da = opcda( localhost , Dummy.Server );
allInfo = propinfo(da)
serverIDInfo = propinfo(da, ServerID )
```

See Also

opchelp

read

Read data synchronously from OPC groups or items

Syntax

```
S = read(GObj)
S = read(Iobj)
S = read(GObj, Source )
S = read(IObj, Source )
```

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. `S` is a structure array containing data for each item in the following fields:

Field Name	Description	Type
`ItemID`	Fully qualified item name	`string`
`Value`	Value	`double`, `string`
`Quality`	Quality of the value	`string`
`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	`string`

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, Source )` and `S = read(IObj, Source )` read data from the source specified by *`Source`* . *`Source`* 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, 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

Configure a client and a 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 )
```

See Also

`readasync` | `refresh` | `write` | `writeasync`

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, device )
```

Examine the event log:

```
pause(2)
disp( Event log: )
showopcevents(da)
```

See Also

`cancelasync` | `read` | `refresh` | `write` | `writeasync`

readAtTime (opchda)

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 string or cell array of strings 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 strings. `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]);
```

See Also

`datenum` | `readRaw` | `readProcessed` | `readModified`

readAtTime (opcua)

Read historical data from nodes of an 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);
```

See Also

`readValue` | `readProcessed` | `readHistory` | `opcuanode`

Introduced in R2015b

readHistory (opcua)

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.

See Also

`opcuanode` | `readAtTime` | `readProcessed` | `readValue`

Introduced in R2015b

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 string containing the item ID for which attributes are requested. `Attribute` is the requested attribute for the item, specified either as a string or 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 stamp(s) 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)
```

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 strings 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.

See Also

`datenum` | `readRaw`

readProcessed

Package: opc.hda

Read server-aggregated data from an OPC HDA server

Syntax

```
DObj =
readProcessed(HdaObj,ItmList,AggregateType,AggregateInterval,StartTime,EndTime
[ItmID,Value,Quality,TimeStamp] =
readProcessed(HdaObj,ItmList,AggregateType,AggregateInterval,StartTime,EndTime
S =
readProcessed(HdaObj,ItmList,AggregateType,AggregateInterval,StartTime,EndTime
```

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 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` 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 strings. `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`
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);
```

See Also

`readRaw` | `readAtTime` | `readModified` | `opc.hdaQualityString`

readProcessed (opcua)

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 string 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.AggregateFnId.

`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

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 ));
```

See Also

Classes

opc.ua.AggregateFnId

Functions

`opcuanode` | `readAtTime` | `readHistory` | `readValue`

Introduced in R2015b

readRaw

Package: opc.hda

Read raw data stored over a time range from HDA server

Syntax

```
DObj = readRaw(HdaClient, ItmList, StartTime, EndTime)
DObj = readRaw(HdaClient, ItmList, StartTime, EndTime, BoundsFlag)
```

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 strings 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, BoundsFlag)` allows you to specify a bounds flag. If `BoundsFlag` is `true`, then the first data point on or outside the defined start and end times will be returned. If `BoundsFlag` is `false`, then only values that were time stamped between `StartTime` (inclusive) and `EndTime` (exclusive) will be included.

Note that one or more time stamps returned for each item may be unique to that item. To retrieve aligned data from an OPC HDA Server, use `readAtTime` or `readProcessed`.

Examples

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 items:

```
DObj = readRaw(hdaObj, { Random.Real8 ,  Random.Real4 }, now-1, now);
```

See Also

datenum | readAtTime | readProcessed | readModified

readValue (opcua)

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 node

This example shows how to read the value from a node identified by its `Index` and `Identifier`.

```
s = opcuaserverinfo( localhost );
UaClient = opcua(s(1));
connect(UaClient);
node = opcuanode(2,10226,UaClient);
[val,tstamp,qual] = readValue(UaClient,node)
```

```
val =
    3.1416

tstamp =
   05-Jun-2015 15:16:15
```

```
qual =
OPC UA Quality ID:
 Good
```

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 values. 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
```

See Also

`browseNamespace` | `getNamespace` | `opcuanode` | `writeValue`

Introduced in R2015b

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)
```

See Also

`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

Connect to the server `Dummy.Server` and remove the public group named `PGroup`:

```
da = opcda( localhost ,  Dummy.Server );
connect(da);
removepublicgroup(da,  PGroup );
```

See Also

`addgroup` | `makepublic`

resample

Class: opc.hda.Data
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 opcdemoHDAData;
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

`interp1` | `tsunion` | `showValues` | `tsintersect`

save

Save OPC Toolbox objects to MAT-file

Syntax

```
save FileName
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 Toolbox objects, `Obj1`, `Obj2`, `...` to the specified MAT-file, `FileName`. If an extension is not specified for `FileName`, then a `.MAT` extension is used.

`save` can be used in the functional form as well as the command form shown above. When using the functional form, you must specify the file name and toolbox objects as 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
```

See Also

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 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 is intended to allow 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's canonical data type:

```
p(1)
```

Read the third property to see the item's quality:

```
p(3)
```

See Also

`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 string to search for, and can contain the wildcard character `*` . `FQID` is a string or cell array of strings. 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 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

Browse Filter	Description
	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 string ( ) 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.* ));
```

See Also

getnamespace | serveritemprops

set

Configure or display OPC Toolbox 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 Toolbox 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 string 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 strings or an empty cell array if the property does not have a finite set of possible string 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 strings, `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 param-value string pairs, structures, and param-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's name:

```
set(grp, Name , myopcgroup );
```

Query the permissible values for the group's `Subscription` property:

```
set(grp, Subscription )
```

More About

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;
```

See Also

`get` | `opchelp` | `propinfo`

showopcevents

Event log summary for OPC Toolbox 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 string, or a cell array of strings 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);
```

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 opcdemoHDAData;
showValues(hdaDataSmall(1))
```

See Also

`disp`

single

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA Data object array to single type matrix

Syntax

```
V = single(DObj)
```

Description

`V = single(DObj)` converts the OPC HDA aata object array `DObj` into a matrix of data type single. `V` is constructed as an `M`-by-`N` array of single values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`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

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 opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vSingle = single(dUnion);
```

See Also

`resample` | `tsintersect` | `tsunion`

stairs

Class: opc.hda.Data
Package: opc.hda

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 opcdemoHDAData;
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 OPC Toolbox software 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
```

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);
```

See Also

`start` | `wait`

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

Property Name	Description	Default
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 strings, each item's plot type is set to the respective plot type in the value array.	`line`
DateFormat	Sets the display format for the *x*-axis of all axes objects into which data is plotted. DateFormat must be one of the date formats recognized by datetick.	`HH:MM:SS`
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]);
```

See Also

`hold` | `datetick`

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 opcdemoHDAData;
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 string 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 opcdemoHDAData;
newObj = tsunion(hdaDataSmall);
```

Find the union using `'hold'` resampling:

```
newObjHold = tsunion(hdaDataSmall, 'hold');
```

See Also

`interp1` | `resample` | `showValues` | `tsintersect`

uint16

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to uint16 matrix

Syntax

```
V = uint16(DObj)
```

Description

`V = uint16(DObj)` converts the OPC HDA data object array `DObj` into an uint16 matrix. `V` is constructed as an `M`-by-`N` array of uint16 values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`DObj` must have the same time stamps for each of the item IDs (elements of `DObj`), othewise 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

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create an uint16 matrix from the result:

```
load opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vUInt16 = uint16(dUnion);
```

See Also

`resample` | `tsintersect` | `tsunion`

uint32

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to uint32 matrix

Syntax

```
V = uint32(DObj)
```

Description

`V = uint32(DObj)` converts the OPC HDA data object array `DObj` into an uint32 matrix. `V` is constructed as an `M`-by-`N` array of uint32 values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`DObj` must have the same time stamps for each of the item IDs (elements of `DObj`), othewise 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

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create an uint32 matrix from the result:

```
load opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vUInt32 = uint32(dUnion);
```

See Also

`resample` | `tsintersect` | `tsunion`

uint64

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to uint64 matrix

Syntax

```
V = uint64(DObj)
```

Description

`V = uint64(DObj)` converts the OPC HDA data object array `DObj` into an uint64 matrix. `V` is constructed as an `M`-by-`N` array of uint64 values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`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

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create an uint64 matrix from the result:

```
load opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vUInt64 = uint64(dUnion);
```

See Also

`resample` | `tsintersect` | `tsunion`

uint8

Class: opc.hda.Data
Package: opc.hda

Convert OPC HDA data object array to uint8 matrix

Syntax

```
V = uint8(DObj)
```

Description

`V = uint8(DObj)` converts the OPC HDA data object array `DObj` into an uint8 matrix. `V` is constructed as an `M`-by-`N` array of uint8 values, where `M` is the number of items in `DObj` and `N` is the number of time stamps in the array.

`DObj` must have the same time stamps for each of the item IDs (elements of `DObj`), othewise 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

Load the OPC HDA example data file, convert the `hdaDataSmall` object to have the same time stamps, and create an uint8 matrix from the result:

```
load opcdemoHDAData;
dUnion = tsunion(hdaDataSmall);
vUInt8 = uint8(dUnion);
```

See Also

`resample` | `tsintersect` | `tsunion`

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.

`wait(GObj, TSec)` will wait at most `TSec` seconds for `GObj` to stop logging. If the group object is still logging when the timeout value is exceeded, an error message is generated.

The `wait` function can be useful when you want to guarantee that data is logged before another task is performed.

You can press `Ctrl+C` to interrupt the `wait` function. An error message will be generated, and control will return to the MATLAB command window.

Examples

Log 60 seconds of data at 1-second intervals from the Matrikon Simulation Server's `Random.Real8` and `Random.UInt4` tags. Display a message indicating that the acquisition is complete, 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);
```

See Also

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.

`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

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 });
```

Read and write values to/from the items:

```
write(grp, {23,  Hello World! })
r = read(grp)
write(itm(1), 15)
r2 = read(itm(1))
```

See Also

`read` | `readasync` | `refresh` | | `writeasync`

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! })
```

See Also

`cancelasync` | `read` | `readasync` | `refresh` | `write`

writeValue (opcua)

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 to be 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's `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.

`writeValue(NodeList,Values)` writes content of `Values`, to the nodes identified by `NodeList`. All nodes must be of the same connected client.

Examples

Write to node

This example shows how to write to a node selected with the name space browser.

```
s = opcuaserverinfo( localhost );
UaClient = opcua(s);
connect(UaClient);
```

```
Node = browseNamespace(UaClient); % Select a variable node
writeValue(UaClient,Node,pi);
```

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

Values specified as a cell array or single value. If writing to a single node, use one value; if writing to an array of nodes, use a cell array of values.

See Also
`browseNamespace` | `getNamespace` | `readValue`

Introduced in R2015b

19

Block Reference

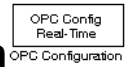
OPC Configuration

Configure OPC clients to use in model, pseudo real-time control options, and behavior in response to OPC errors and events

Library

OPC Toolbox

Description

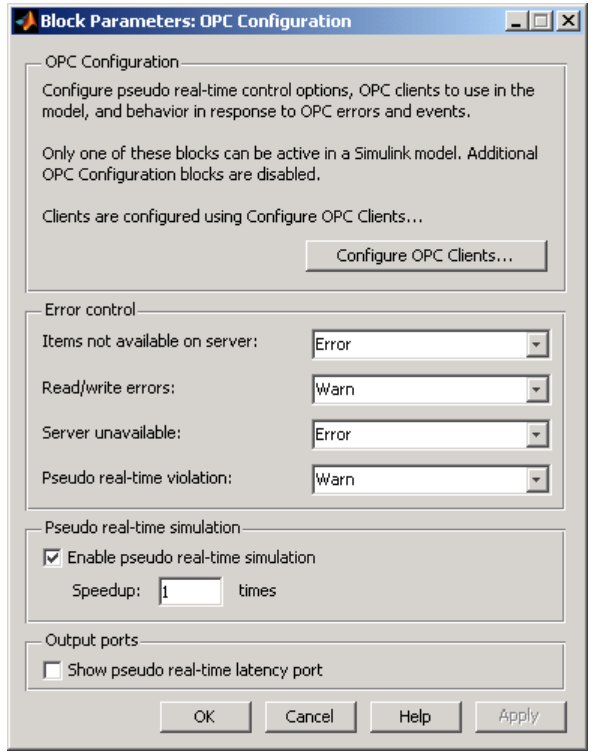


The OPC Configuration block defines the OPC clients to be used in a model, configures pseudo real-time behavior for the model, and defines behavior for OPC errors and events.

The block has no input ports. One optional output port displays the model latency (time spent waiting in each simulation step to achieve pseudo real-time behavior).

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 becomes disabled.

Dialog Box



Configure OPC Clients

Opens the OPC Client Manager for this 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 server. See “Use the OPC Client Manager” on page 10-17 for more information.

Error control

Defines actions that Simulink software must take 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. The following table describes each error or event.

Error/Event	Description	Default
Items not available on server	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.	`error`
Read/write errors	Defines the behavior when a read or write operation fails.	`warn`
Server unavailable	Defines the behavior when the client cannot connect to the OPC server, or when the server sends a shutdown event to the client.	`error`
Pseudo real-time violation	Defines the behavior when the simulation runs slower than real time. See the **Pseudo real-time simulation** options for more information.	`warn`

Pseudo real-time simulation

Allows you to configure options for running the simulation in pseudo real time. When **Enable pseudo real-time simulation** is checked, the model execution time matches the system clock as closely as possible by slowing down the simulation appropriately. 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.

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. You can also output the model latency using the **Show pseudo real-time latency port** setting.

Show pseudo real-time latency port

When checked, the pseudo real-time latency (in seconds) is output from the block. Pseudo real-time latency is the time spent waiting for the system clock during each step. If this value is negative, the simulation runs slower than real time, and the behavior defined in the **Pseudo real-time violation** setting determines the action that Simulink takes.

See Also

OPC Read, OPC Write

OPC Quality Parts

Convert OPC quality ID into vendor, major, minor, and limit status

Library

OPC Toolbox

Description

The OPC Quality Parts block converts an OPC quality ID vector into four parts:

•

- Vendor status
- Major quality
- Quality substatus
- Limit status.

The Quality port of an OPC Read block generates quality IDs.

For more information on quality parts, see “OPC Quality Strings” on page A-2.

See Also

OPC Read

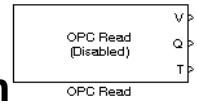
OPC Read

Read data from OPC server

Library

OPC Toolbox

Description



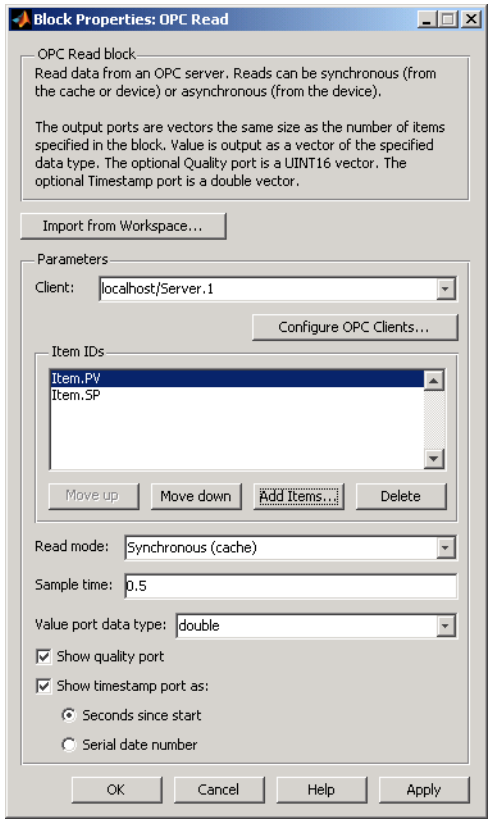
The OPC Read block reads data from one or more items on an OPC server. The read operation takes place synchronously (from the cache or from the device) or asynchronously (from the device).

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 additional outputs. The time stamp may 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 dialog without first including an OPC Configuration block in the model.

Dialog Box



Import from Workspace

Allows you to import settings for the OPC Read block from a `dagroup` object in the 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

Defines the OPC client associated with this block. You can add additional clients to the list using **Configure OPC Clients**. For more information, see "Use the OPC Client Manager" on page 10-17.

Item IDs

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 reorder 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

Defines the read mode for this block. Available options are `Asynchronous`, `Synchronous (cache)`, or `Synchronous (device)`. Synchronous reads have slightly more overhead than asynchronous reads, but they are generally more reliable than asynchronous reads.

Sample time

Defines the sample time for the block. 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

Defines the data type for the value output. The OPC server is responsible for converting all data to the required type.

Note For items with a Canonical Data Type of `logical`, the OPC Read block outputs `-1` for signed data types, or the maximum value for unsigned data types, when the item value is "true". A value of `0` is output when the item value is "false".

Show quality port

When checked, the quality IDs of all the items are output in the second port as a vector of unsigned 16-bit integers. Use the OPC Quality Parts block to separate the quality ID into component parts.

Show timestamp port

When checked, the timestamps for each of the items are output in the last port as a vector of doubles. You choose whether to output the timestamps as **Seconds since start** (i.e., simulation time) or as **Serial date numbers** (i.e., real-world time).

See Also

OPC Configuration, OPC Quality Parts, OPC Write

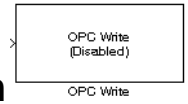
OPC Write

Write data to OPC server

Library

OPC Toolbox

Description

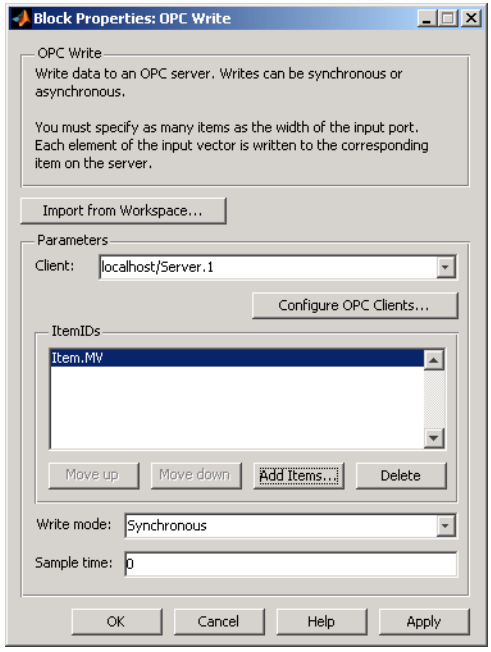


The OPC Write block writes data to one or more items on an OPC server. The write operation takes place synchronously or asynchronously.

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.

Dialog Box



Import from Workspace

Allows you to import settings for the OPC Write block from a `dagroup` object in the base workspace. The client, item IDs, and sample time are updated based on the properties of the imported group.

Client

Defines the OPC 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 10-17.

ItemIDs

Shows the items to be written to the specified server. You can add items to the list using **Add Items**, or delete items using **Delete**. You can reorder the items in the list using **Move Up** or **Move Down**. Each element of the input port is written to the corresponding item in the list.

Write mode

Defines the write mode for this block. Available options are `Asynchronous` and `Synchronous`. Synchronous writes have slightly more overhead than asynchronous writes, but they are generally more reliable than asynchronous writes.

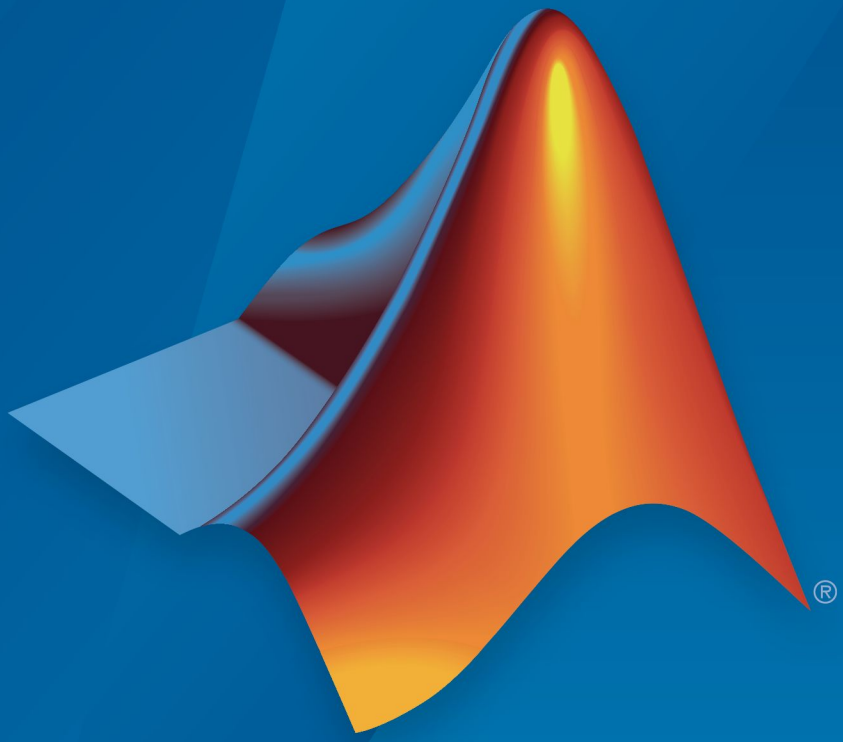
Sample time

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.

See Also

OPC Configuration, OPC Read

OPC Toolbox™ Release Notes



MATLAB® & SIMULINK®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

OPC Toolbox™ Release Notes

Contents

R2014a

R2013b

R2013a

R2012b

R2012a

R2011b

R2011a

R2010b

Bug Fixes

R2010a

Bug Fixes

R2009b

Bug Fixes

R2009a

R2008b

R2008a

R2007b

R2007a

R2006b

R2006a

R14SP3

R14SP2+

R2016a

Version: 4.0.1

Bug Fixes

R2015b

Version: 4.0

New Features

Bug Fixes

Support for OPC-UA standard

This release of OPC Toolbox™ includes support for the OPC Unified Architecture (UA) standard protocol.

For information on getting started with OPC UA servers, nodes, and clients, see Unified Architecture.

Note OPC UA support is available only for 64-bit Windows® operating systems.

R2015a

Version: 3.3.3

Bug Fixes

R2014b

Version: 3.3.2

Bug Fixes

R2014a

Version: 3.3.1

New Features

Bug Fixes

OPC Client App Renamed OPC Data Access Explorer

The app known as OPC Client in past releases is now called the OPC Data Access Explorer. opcDataAccessExplorer is a new corresponding command-line function that opens the app programmatically. The function `opctool` continues to open this app.

R2013b

Version: 3.3

Bug Fixes

R2013a

Version: 3.2

New Features

Bug Fixes

Namespace browser for viewing item names on OPC HDA and OPC DA servers

There are new tools that let you graphically browse the name space for OPC HDA or OPC DA servers:

- For more information on the OPC DA namespace browser, see the browsenamespace reference page.
- For more information on the OPC HDA namespace browser, see the browseNameSpace reference page.

opchda function for creating an OPC HDA client object

A new function, `opchda`, creates an OPC HDA client object in MATLAB. For more information on this function, see the opchda reference page.

New and enhanced troubleshooting utilities

The troubleshooting utility opcsupport has been enhanced to support OPC HDA as well as OPC DA functionality.

Two new troubleshooting utilities are provided specifically for OPC HDA and OPC DA:

- For more information on the OPC DA troubleshooting utility, see the opc.daSupport reference page.
- For more information on the OPC HDA troubleshooting utility, see the opc.hdaSupport reference page.

R2012b

Version: 3.1.2

Bug Fixes

R2012a

Conversion of Error and Warning Message Identifiers

For R2012a, error and warning message identifiers have changed in OPC Toolbox.

Compatibility Considerations

If you have scripts or functions that use message identifiers that changed, you must update the code to use the new identifiers. Typically, message identifiers are used to turn off specific warning messages, or in code that uses a try/catch statement and performs an action based on a specific error identifier.

To determine the identifier for a warning, run the following command just after you see the warning:

```
[MSG,MSGID] = lastwarn;
```

This command saves the message identifier to the variable `MSGID`.

To determine the identifier for an error, run the following commands just after you see the error:

```
exception = MException.last;
MSGID = exception.identifier;
```

Tip Warning messages indicate a potential issue with your code. While you can turn off a warning, a suggested alternative is to change your code so that it runs without warnings.

R2011b

Version: 3.1

New Features

Bug Fixes

64-Bit Support

This release supports OPC Toolbox in 64-bit MATLAB®.

Before using OPC Toolbox with 64-bit MATLAB for the first time, you must run opcregister to install the 64-bit OPC Core Components. This applies even if you have previously installed the OPC Core Components, either through 32-bit MATLAB or installation of an OPC Server on the local host. Failure to install the 64-bit OPC Core Components that ship with this version of OPC Toolbox can result in local OPC servers being unavailable to OPC Toolbox.

R2011a

Version: 3.0

New Features

Bug Fixes

Support for Historical Data Access Servers

OPC Toolbox now supports OPC Historical Data Access (OPC HDA) specification version 1.2. This allows you to access and visualize historical process data from an OPC Historical Data Access server (commonly referred to as an historian).

Key features of OPC HDA support are:

- You can browse networks for Historical Data Access servers.
- A single OPC HDA client object allows you to manage connections to an HDA server, to browse the server name space, and to read raw and processed data from the server.
- An OPC Historical Data Access object allows you to easily manipulate and visualize data retrieved from OPC HDA servers.

New OPC HDA Functions

The OPC HDA read functions include:

- readRaw — Reads raw historical data for a selection of item tags over a selected time span.
- readAtTime — Reads data from the server at specific time intervals. It can return array data as an output, by specifying an optional `datatype` argument.
- readProcessed — Reads processed data from the server for chosen item tags using an aggregate type defined from a list of available types. It can return array data as an output, by specifying an optional `datatype` argument.
- readModified — Reads all modified data between two times for a particular item.

The OPC HDA data object functions include:

- tsunion — (Time stamp union) Converts all data to have the same time stamps, by resampling at the union of the time stamps of all items.
- tsintersect — (Time stamp intersection) Converts all data to have the same time stamps, using the time stamps common to all items in the array. Removes all data that is not common to all elements (items) in the data object.
- resample — Converts all data to have the given (regularly) sampled time stamps that you provided. Combines elements with the same item ID, so that `resample` creates data objects with unique item IDs.
- plot — Generates a plot so you can visualize the historical data.

Getting Help

For a command-window listing of available functions and links to their specific help, type:

```
help opc.hda
```

You access the Historical Data Access (HDA) functions through the `opc.hda` package. For this reason, to get help on a particular HDA function, you must prefix the function name with `opc.hda`. For example, to get help on the `getServerInfo` function, type:

```
help opc.hda.getServerInfo
```

Other prefixes, such as `opc`, `opc.hda.Data`, or `opc.hda.Client` might be necessary, as listed in the output for `help opc.hda`.

To view OPC HDA documentation in the help browser, see

- Quick Start: Using OPC Historical Data Access Functions
- Historical Data Access User's Guide
- OPC Historical Data Access (HDA) functions category

Demos

Demos for OPC Historical Data Access Tutorials are in the help documentation under the `Demos` node in OPC Toolbox.

R2010b

Version: 2.1.6

Bug Fixes

R2010a

Version: 2.1.5

Bug Fixes

R2009b

Version: 2.1.4

Bug Fixes

R2009a

Version: 2.1.3

Bug Fixes

R2008b

Version: 2.1.2

Bug Fixes

R2008a

Version: 2.1.1

Bug Fixes

R2007b

Version: 2.1

New Features

Bug Fixes

Graphical Display of OPC Data

The new trend function provides a graphical display of live OPC data for an OPC group object. You can now watch live data stream to a MATLAB figure window. The `trend` functionality is independent of OPC logging tasks. More information is available on the `trend` reference page.

Time Series Objects Now Supported

OPC Toolbox software now supports time series objects, allowing easier analysis and visualization of time domain data in MATLAB software. The addition of this functionality involves the extension of the existing OPC Toolbox functions getdata and opcread to support the creation of MATLAB time series objects. The new functions opcstruct2timeseries and opcqid have also been added to the toolbox to support this feature. More information is available on the reference pages for these functions.

R2007a

Version: 2.0.4

Bug Fixes

R2006b

Version: 2.0.3

Bug Fixes

R2006a

Version: 2.0.2

New Features

opcregister Function Enhanced

The opcregister function has been enhanced with a `-silent` option to install OPC Foundation Core components without dialog boxes.

R14SP3

Version: 2.0.1

New Features

Full Event Log

The event log is no longer cleared when a new event arrives and the event log is full. Instead, the oldest event is removed to make space for the new event. For more information, see the reference page for the EventLogMax property, by typing

```
doc EventLogMax
```

Changing TimerPeriod Property

You can now change the TimerPeriod property while a client object is connected.

R14SP2+

Version: 2.0

New Features

Support for Simulink Software: OPC Toolbox Block Library

This release of OPC Toolbox software includes support for communicating with OPC servers from Simulink® software. The OPC Toolbox block library includes blocks for reading data from an OPC server, writing data to an OPC server, and running a Simulink simulation in pseudo real time (by slowing the simulation to run at system time). For more information on using OPC Toolbox software with Simulink software, see Using the OPC Toolbox Block Library in the User's Guide.

New Item Property: QualityID

The QualityID property has been added to `daitem` objects. The `QualityID`, expressed as a 16-bit unsigned integer, represents the quality of the data item when last read. To work with the `QualityID` property, you use the function opcqparts to convert the `QualityID` property into vendor, major, substatus, and limit status information; and the function opcqstr to convert the `QualityID` property into a string. For more information, see the `QualityID` property reference page.