



# Applied AutoMod® 12.3

## Model Communications User's Guide

**think it. apply it.**

## **Copyright**

© 2009 Applied Materials, Inc.  
3050 Bowers Avenue  
Santa Clara CA 95054  
U.S. and Foreign Patents Pending

All rights reserved. No part of this book may be reproduced in any form without written permission from Applied Materials, Inc.

Published August 2009

## **Trademarks**

Applied Materials and the Applied Materials logo are registered trademarks of Applied Materials, Inc. in the U.S. and other countries.

AutoMod is a registered trademarks of Applied Materials, Inc. in the U.S. and/or other countries.

AutoStat and AutoView are trademarks of Applied Materials, Inc. in the U.S. and/or other countries.

All other trademarks and registered trademarks contained herein are the property of their respective owners.

## **Confidentiality Notice**

The materials and information contained herein are being provided by Applied Materials to its Customer solely for the Customer's use for its internal business purposes. Applied Materials retains all right, title, interest in, and copyrights to the materials and information herein. The materials and information contained herein constitute confidential information of Applied Materials, and the Customer shall not disclose or transfer any of these materials or information to any third party.

## **U.S. Government Restricted Rights**

The software and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the government are subject to the restrictions set forth in subdivision (c)(1)(ii) of the Rights of Technical Data and Computer Software clause at DFARS 252.227-7023, and in subdivision (c)(1) and (2) of the Commercial Computer Software-Restricted Rights clause at 48 CFR 52.27-19, as applicable. The contractor/manufacturer is Applied Materials, Inc., 3050 Bowers Avenue, Santa Clara, CA 95054.

## **Limited Warranty**

Applied Materials makes no warranties, express or implied, including without limitation, any warranties of merchantability or fitness for a particular purpose, regarding the licensed programs or specifications or their performance, except as expressly provided in the Applied Materials software license agreement signed by both Applied Materials and the Customer.

In no event shall Applied Materials be liable for any incidental, special, consequential, or exemplary damages including, but not limited to, loss of anticipated profits or benefits resulting from any defect in the software or its documentation except as expressly provided in the Applied Materials software license agreement.

The information in this document is subject to change without notice. Any sample code that appears in this document and that may be included with the software product is included for illustration only and is, therefore, unsupported.

# Contents

|                  |   |            |
|------------------|---|------------|
|                  | <b>About This Book</b>  | <b>v</b>   |
|                  | Purpose   | v          |
|                  | Audience  | v          |
|                  | Prerequisites   | v          |
|                  | Organization  | v          |
|                  | Conventions   | vii        |
|                  | Additional Information  | viii       |
| <b>Chapter 1</b> | <b>Introduction to Model Communications</b>                   | <b>1.1</b> |
|                  | Why use model communications?                                 | 1.1        |
|                  | Emulating an automated material handling system (AMHS)        | 1.1        |
|                  | Developing modular simulations                                | 1.2        |
|                  | Model communications overview                                 | 1.2        |
|                  | The Model Communications module                               | 1.2        |
|                  | The Model Communications Plus module                          | 1.4        |
| <b>Chapter 2</b> | <b>Communicating with an OPC Server</b>                       | <b>2.1</b> |
|                  | Introduction  | 2.1        |
|                  | What is an OPC server?  | 2.3        |
|                  | Item properties   | 2.5        |
|                  | Group properties  | 2.6        |
|                  | Emulation versus simulation                                   | 2.8        |
|                  | Designing a simulation model for use in emulation             | 2.8        |
|                  | Managing connections with OPC servers                         | 2.9        |
|                  | Opening connections   | 2.9        |
|                  | Verifying connections   | 2.22       |
|                  | Closing connections   | 2.25       |
|                  | Writing item values   | 2.25       |
|                  | Writing item values synchronously with validation             | 2.26       |
|                  | Writing item values synchronously without validation          | 2.28       |
|                  | Reading item values   | 2.30       |
|                  | Reading an item's value at a specific time during a model run | 2.31       |
|                  | Subscribing to groups and items                               | 2.33       |
|                  | Refreshing item subscriptions                                 | 2.35       |
|                  | Responding to changes in subscribed item values               | 2.37       |
|                  | Running the model   | 2.40       |
|                  | Synchronizing a model's runtime rate with a control system    | 2.41       |
|                  | Signaling a control system when paused or continuing          | 2.43       |
|                  | Creating delays measured in real time                         | 2.44       |
|                  | Calculating the real time that passes between model events    | 2.45       |

|                   |   |            |
|-------------------|---|------------|
|                   | Viewing statistics and communication information after a run . . . . .          | 2.45       |
|                   | Displaying item values and update statistics . . . . .                          | 2.46       |
|                   | Using trace files . . . . .   | 2.48       |
|                   | Testing scenarios . . . . .   | 2.55       |
| <b>Chapter 3</b>  | <b>Socket Communications . . . . .</b>  | <b>3.1</b> |
|                   | Introduction . . . . .  | 3.1        |
|                   | Managing socket connections . . . . .   | 3.1        |
|                   | Opening a connection . . . . .  | 3.2        |
|                   | Verifying a connection . . . . .  | 3.11       |
|                   | Closing a connection . . . . .  | 3.13       |
|                   | Sending and reading messages . . . . .  | 3.13       |
|                   | Message formats . . . . .   | 3.13       |
|                   | Sending a message . . . . .   | 3.16       |
|                   | Reading a message . . . . .   | 3.19       |
|                   | Interpreting a message . . . . .  | 3.23       |
|                   | Synchronizing two models without the MMS Server . . . . .                       | 3.26       |
| <b>Chapter 4</b>  | <b>Multi-Model Synchronization (Model Communications Plus module) . . . . .</b> | <b>4.1</b> |
|                   | Introduction . . . . .  | 4.1        |
|                   | Understanding event-driven simulation . . . . .                                 | 4.2        |
|                   | Synchronizing simulation clocks . . . . .                                       | 4.4        |
|                   | Communicating between synchronized models . . . . .                             | 4.7        |
|                   | Sending an MMS message . . . . .  | 4.7        |
|                   | Reading an MMS message . . . . .  | 4.13       |
|                   | Interpreting an MMS message . . . . .   | 4.17       |
|                   | Starting a synchronized simulation . . . . .                                    | 4.18       |
|                   | Setting up remote machines . . . . .  | 4.18       |
|                   | Defining the model data file . . . . .  | 4.19       |
|                   | Starting the MMS Server . . . . .   | 4.22       |
|                   | Using trace files . . . . .   | 4.25       |
|                   | Trace file format . . . . .   | 4.25       |
|                   | Running a model with a trace file . . . . .                                     | 4.28       |
| <b>Appendix A</b> | <b>OPC Quality Codes . . . . .</b>  | <b>A.1</b> |
|                   | Codes –1 through 28 (“Bad” or unreliable data) . . . . .                        | A.1        |
|                   | Codes 64 through 91 (“Uncertain” data) . . . . .                                | A.2        |
|                   | Codes 192 through 219 (“Good” or reliable data) . . . . .                       | A.3        |
| <b>Appendix B</b> | <b>OPC Communication Functions . . . . .</b>                                    | <b>B.1</b> |
|                   | System functions . . . . .  | B.1        |
|                   | MilliSleep function . . . . .   | B.1        |
|                   | OPCAddGroup function . . . . .  | B.2        |
|                   | OPCAddItem function . . . . .   | B.3        |
|                   | OPCConnectServer function . . . . .   | B.5        |
|                   | OPCDisconnect function . . . . .  | B.6        |
|                   | OPCFlushWrite function . . . . .  | B.7        |
|                   | OPCInit function . . . . .  | B.8        |
|                   | OPCListConnections function . . . . .   | B.9        |

|                                   |      |
|-----------------------------------|------|
| OPCListItems function             | B.10 |
| OPCReadInteger function           | B.11 |
| OPCReadReal function              | B.11 |
| OPCReadString function            | B.12 |
| OPCRefreshSubscription function   | B.13 |
| OPCSubscribe function             | B.14 |
| OPCWriteInteger function          | B.16 |
| OPCWriteIntegerWait function      | B.17 |
| OPCWriteReal function             | B.18 |
| OPCWriteRealWait function         | B.20 |
| OPCWriteString function           | B.21 |
| OPCWriteStringWait function       | B.22 |
| User-defined OPC functions        | B.23 |
| opccallback function              | B.23 |
| Defining a reference to a machine | B.26 |
| Defining a reference to an item   | B.26 |

## **Appendix C    Socket Communication Functions ..... C.1**

|                                |      |
|--------------------------------|------|
| System functions               | C.1  |
| AcceptSocketPort function      | C.1  |
| AcceptSocketService function   | C.2  |
| CloseSocket function           | C.3  |
| ConnectSocketPort function     | C.3  |
| ConnectSocketService function  | C.4  |
| FEvFirstTime function          | C.5  |
| GetSocketName function         | C.5  |
| GetSocketHost function         | C.5  |
| GetSocketNum function          | C.6  |
| MachineName function           | C.6  |
| NetworkConvertShort function   | C.7  |
| NetworkConvertInteger function | C.7  |
| NetworkConvertFloat function   | C.8  |
| NetworkConvertDouble function  | C.8  |
| ReadSocket function            | C.9  |
| ReadSocketString function      | C.9  |
| SendSocket function            | C.10 |
| SendSocketString function      | C.11 |
| SetAcceptBlocking function     | C.11 |
| SetAcceptTimeOut function      | C.12 |
| SetConnectBlocking function    | C.13 |
| SetConnectTimeOut function     | C.14 |
| SetNumAcceptsPort function     | C.14 |
| SetNumAcceptsService function  | C.15 |
| SetSocketMessages function     | C.16 |
| SetSocketNonBlocking function  | C.16 |
| SetSocketTimeOut function      | C.17 |
| ValidSocket function           | C.18 |
| User-defined socket functions  | C.18 |
| model nextclock function       | C.19 |

## **Appendix D    Multi-Model Synchronization Functions ..... D.1**

|  |            |
|--|------------|
| System functions                       | D.1        |
| MMSyncReadMessageInteger function      | D.1        |
| MMSyncReadMessageNumIntegers function  | D.2        |
| MMSyncReadMessageNumReals function     | D.3        |
| MMSyncReadMessageNumStrings function   | D.3        |
| MMSyncReadMessagePrint function        | D.4        |
| MMSyncReadMessageReal function         | D.4        |
| MMSyncReadMessageString function       | D.5        |
| MMSyncReadMessageStringLength function | D.6        |
| MMSyncReadMessageTime function         | D.7        |
| MMSyncReadMessageType function         | D.7        |
| MMSyncSendMessage function             | D.8        |
| MMSyncSendMessageClear function        | D.8        |
| MMSyncSendMessageInteger function      | D.9        |
| MMSyncSendMessageModel function        | D.10       |
| MMSyncSendMessageNumDatas function     | D.11       |
| MMSyncSendMessageNumIntegers function  | D.12       |
| MMSyncSendMessageNumReals function     | D.13       |
| MMSyncSendMessageNumStrings function   | D.14       |
| MMSyncSendMessagePrint function        | D.15       |
| MMSyncSendMessageReal function         | D.16       |
| MMSyncSendMessageString function       | D.16       |
| MMSyncSendMessageType function         | D.17       |
| User-defined MMS functions             | D.18       |
| model mmsyncmessage function           | D.18       |
| <b>Glossary</b>                        | <b>G.1</b> |
| <b>Index</b>                           | <b>I.1</b> |

# About This Book

To help you find the right information about using the AutoMod software, this section briefly describes the purpose, intended audience, prerequisites, organization, and typographic conventions used in this guide. It also lists additional documentation that can help you learn more about AutoMod products.

---

## Purpose

This guide explains how to run models as a combined simulation on networked machines, control models from third-party applications, and communicate with an OPC server to test a control system by replacing system hardware with an AutoMod simulation.

---

## Audience

This guide is designed for simulation practitioners who want to build models that communicate with other applications during simulation.

---

## Prerequisites

This book assumes you already know how to build, run, and analyze a model in the AutoMod software. For more information about these tasks, refer to the *AutoMod User's Guide*.

---

## Organization

This book contains the following chapters:

| Chapter  | Purpose  |
|--|--|
| Chapter 1,<br>"Introduction to Model Communications" | Outlines the reasons for using model communications, the types of communication that are supported, and the difference between the Model Communications module and the Model Communications Plus module. |

| Chapter   | Purpose  |
|---|--|
| Chapter 2, "Communicating with an OPC Server"                               | Explains how to build a model that emulates an automated material handling system by communicating with an OPC server throughout a simulation.                           |
| Chapter 3, "Model Communications Module"                                    | Explains how to build a model that communicates with another model or a third-party application through socket connections.  |
| Chapter 4, "Multi-Model Synchronization (Model Communications Plus module)" | Explains how to build two or more models that communicate through socket connections. Describes how to synchronize the models by connecting to a synchronization server. |
| Appendix A, "Quality Codes"   | Describes the OPC quality codes that OPC servers provide for each item value that is read by a client application.   |
| Appendix B, "OPC Communications Functions"                                  | Summarizes the functions you can use in a model to communicate with an OPC server.   |
| Appendix C, "Socket Communications Functions"                               | Summarizes the functions you can use in a model to communicate with another model or a third-party application through socket connections.                               |
| Appendix D, "Multi-Model Synchronization Functions"                         | Summarizes the functions you can use to communicate with multiple models through a synchronization server.   |
| Glossary  | Provides definitions of various terms used throughout the Model Communications User's Guide.   |



---

## Conventions

This guide uses the following conventions:

**Bold** text indicates:

- Input you can type using the keyboard.
- Names of dialog box entities.

The monospace font indicates:

- Input you can type using the keyboard.
- AutoMod syntax (function names, actions, keywords, and so on).
- Text that the software displays.
- The contents of a file.

<>

The left and right angle brackets (<>) indicate that you substitute the information requested by the word or words enclosed by the brackets. For example, in the following command, you type the <model\_data\_file\_path> appropriate for your system.

```
MultiModelSync -me <model_data_file_path>
```

If the path to the model data file on your machine is C:\Models\Mymodel\Data.txt, the command is

```
MultiModelSync -me C:\Models\Mymodel\Data.txt
```

---

## Additional Information

Use the following sources for obtaining more information about Applied Materials products:

| Reference                           | Purpose   |
|-------------------------------------|---|
| <i>AutoMod User's Guide</i>         | Introduces a modeling methodology and discusses how to use the software's graphical interface. Explains a model's process system and its components and describes how to create business graphs of simulation statistics and how to print or plot a view of your model's graphics. Includes instructions for running a simulation, analyzing the model results, and debugging model errors. Explains how to design movement systems in the AutoMod software, create model graphics in ACE, animate model graphics with kinematics, create movies of a simulation using AutoView, and conduct model experiments and analysis using AutoStat. |
| AutoMod Help                        | Explains how to write model logic and use the AutoMod editor. The online help also discusses how to use AutoMod utilities including AutoStat, AutoView, the Model Zip Archiver, the OPC Utility, the Process Server, and the Web tools. The help includes information on how to create a custom model interface using ActiveX controls and provides a reference library for previewing and selecting graphics from the graphic templates installed with the AutoMod software.   |
| AutoMod Tutorials                   | Provides instructions for building a model in AutoMod, designing a vehicle graphic in ACE, and animating vehicle and robot graphics using kinematics.   |
| <i>Getting Started with AutoMod</i> | Teaches how to use the AutoMod and AutoStat software through examples and chapter exercises.  |
| <i>AutoMod Runtime User's Guide</i> | Describes how to run an existing simulation in the AutoMod software and experiment and analyze the simulation using AutoStat.   |



# Introduction to Model Communications

This chapter, you will learn:

- Uses for model communications
- The communication types supported by the Model Communications module
- How models are synchronized using the Model Communications Plus module

---

## Why use model communications?

The Model Communications module (MCM) is an enhancement to AutoMod that allows a model to communicate over a network with other software applications (for example, other AutoMod models or industrial control systems). Using the model communications module, you can:

- Test control systems by emulating an automated material handling system (AMHS).
- Develop modular simulations composed of multiple models running as an integrated system

These concepts are discussed in greater detail in the following sections.

## Emulating an automated material handling system (AMHS)

*Emulation* is a process that allows you to test a control system by using an AutoMod model to act in place of the physical material handling hardware. The goal of an emulation model is to provide the same responses to a control system as would be provided by the physical system. While the model is running, the control system cannot tell the difference between the real system and the AutoMod model.

Using emulation accelerates the commissioning of a new material handling system by allowing you to test the control system before hardware installation is complete. Emulation also allows you to design and test a broad range of system conditions, which can help you identify operation problems that otherwise might not have surfaced until the real system was ramped and in use for some time. Using the

AutoStat software, you can run your scenarios on multiple machines simultaneously, which can reduce the time required to test the system.

Emulation can also be a powerful training tool. You can use an emulation model to train personnel in a simulated environment that uses the same interface as the real system. Employees trained using an emulation model can observe how their actions affect a system in a way that is physically impossible in most real systems.

## Developing modular simulations

The Model Communications module allows you to design a simulation that consists of two or more models that can share information over a network. You can design the models to function as independent components that can either be run as separate simulations or as a synchronous, unified system.

Developing a modular simulation is useful for dividing large simulation projects among multiple model builders. After agreeing on the type and format of messages to be sent between models, the builders can separately design pieces of a model. Once completed, the pieces can run synchronously to simulate a single system.

Modular simulations are also useful when you want to separately simulate individual pieces of a system. For example, you can analyze an automotive paint shop in one model and an assembly line in another model, or you can combine the two models to analyze how changes in the paint shop affect assembly.

---

## Model communications overview

There are two communications modules that you can purchase to send and read data in a model:

- The Model Communications module
- The Model Communications Plus module

A general description of each module is provided in the following sections.

## The Model Communications module

The Model Communications module allows models to communicate by calling functions to send and read data through a network connection. The functions that you call in a model to manage connections and

read and write data depend on the type of communication you are using. The Model Communications module supports the following types of communication:

- Communication with an OPC server
- Communication using sockets

### Communicating with an OPC server

OLE for Process Control (OPC) is a Windows-based client/server technology that allows software applications and physical devices to share data without the need for proprietary communications drivers. OPC servers are used in manufacturing systems to provide a means for interoperability among automation and control systems, field devices, and business software applications.

To emulate a manufacturing system, an AutoMod model can communicate with an OPC server in place of the devices in the physical system. For more information, see Chapter 2, “Communicating with an OPC Server.”

### Communicating using sockets

A *socket* is a software object that connects a software application to a network protocol, such as TCP/IP. The Model Communications module supports socket communication for passing string messages between one or more models during a simulation. If you are familiar with C code, you can also define structures to send and receive multiple types of data in the same message (for example, string, integer, real, and so on). For more information, see Chapter 3, “Socket Communications.”

---

**Note** Because sockets are supported by both the Windows and Unix operating systems, you can use socket connections to communicate with a control system running on a Unix machine.

---

Using the Model Communications module, you can also write code to synchronize two models. *Synchronization* is the process of ensuring that the simulation clock advances in multiple models at the same rate. The models are synchronized by alternately causing one model to delay while the other model advances. For easier synchronization, or to synchronize more than two models, use the Model Communications Plus module.

## The Model Communications Plus module

The Model Communications Plus module includes all of the features of the Model Communications module, as well as additional features designed for multi-model communication, including:

- **The Multi-Model Synchronization Server (MMS Server)** - The MMS Server automatically synchronizes all connected models in the simulation. The MMS Server also automatically opens connections to participating models and allows you to easily send and read messages containing multiple types of data.
- **Five runtime licenses** - These licenses allow you to run and synchronize five models simultaneously. The number of models that you can synchronize with the Model Communications Plus module is limited only by the number of runtime licenses that you own.

To synchronize models using the MMS Server, you do not need to write any synchronization code, you only need to create a tab-delimited text file that defines the location of the models that are participating in the simulation. When you start the server, it automatically opens a socket connection to each model and synchronizes the models throughout the simulation. The MMS Server allows you to choose between two types of synchronization: exact synchronization or synchronization at periodic intervals for increased performance. For more information about the Model Communications Plus module, see Chapter 4, “Multi-Model Synchronization (Model Communications Plus module).”



# Communicating with an OPC Server

In this chapter, you will learn:

- The difference between an OPC server, groups, and items
- The difference between emulation models and simulation models
- How to manage connections with OPC servers
- How to write item values to a server
- How to read item values from a server
- How to synchronize an emulation model's runtime rate with a control system
- How to display item values and statistics during a model run
- How to use trace files

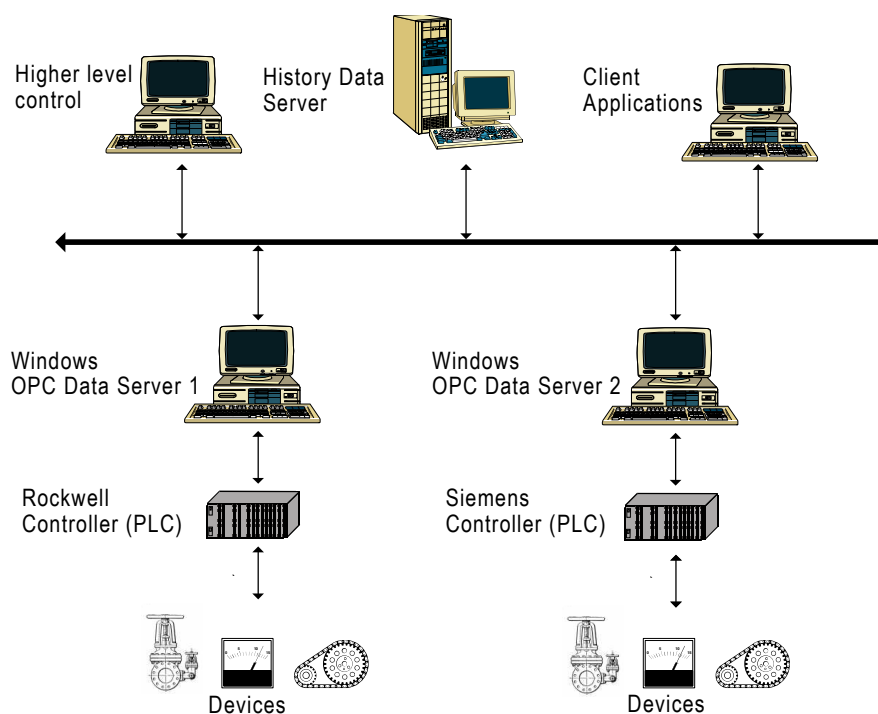
---

## Introduction

Devices in a material handling system are controlled by programmable logic controllers (PLCs). PLCs continuously scan data from sensors in the system, such as presence sensors, temperature sensors, pressure sensors, optical sensors, and so on. PLCs contain a control system that performs logic based on the scanned data and then writes output to system devices (for example, motors, lights, and valves) to control the devices based on current system conditions.

OLE for Process Control (OPC) is a client/server technology that provides a standard interface for sharing data between a system's industrial devices and various software applications, without the need for proprietary drivers. In a system that uses OPC technology, one or more OPC data servers read and write values to PLCs. The data

servers provide networked software applications with access to each PLC's control system and to device information. This system architecture is illustrated below:



To emulate the devices in the manufacturing system, an AutoMod model connects to one or more OPC servers as a client application. OPC servers can read and write values to a controller, in the same way as system devices. Consequently, the AutoMod model can emulate the devices in the system by reading and writing values to the OPC server, in place of the system devices.

Typically, devices react to state changes as directed by a controller; for example, a controller might send a message containing values that cause a motor to change its section velocity or stop or start. Devices send messages to the controller to indicate the status of the device, configuration parameters, and so on.

To accurately emulate the devices in the system, you must first gather detailed information about the values that are passed between the devices and the PLC. You can then design an emulation model to replace the device by reading and writing the same values to the OPC server during a model run. Because the model provides the same responses to the control system as the real system hardware would, the control system cannot tell the difference between the real system and the AutoMod model.



---

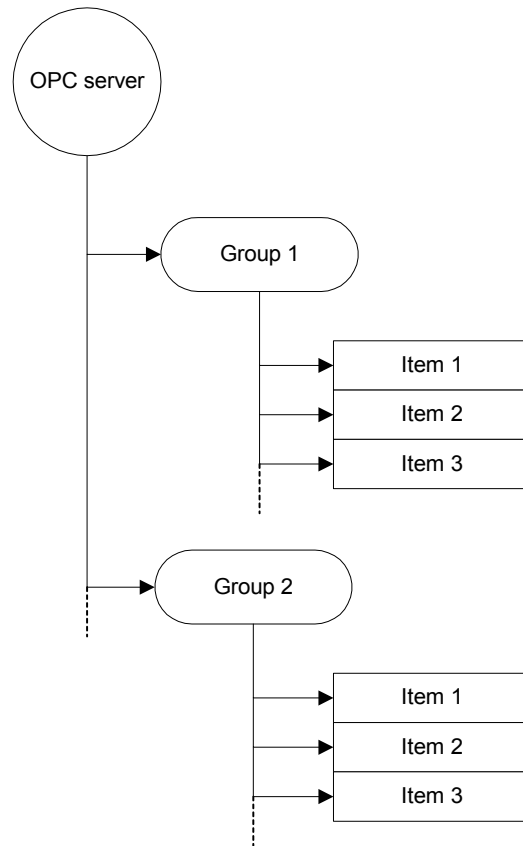
## What is an OPC server?

An *OPC server* is a software application running on the Windows platform that provides a standard interface for OPC clients (other software applications) to access data from a device. An OPC server contains two types of objects that can be used to access and organize data: items and groups.

An *item* is a reference to a data value that is stored on the OPC server. A unique item is created on the server for every data value that is accessible by client applications. An item is similar to a variable in AutoMod; it is a name that is associated with a stored value. Like variables, items differ by the type of data to which they refer; for example, numeric or string values. The server maintains information about each item, such as the item's type, its current value, the time it was last changed, and so on. For more information, see “Item properties” on [page 2.5](#).

A *group* is a client-specific collection of items. Each client application that connects to an OPC server can create its own groups for organizing the items on the server. Each group can consist of one or more items. The server independently keeps track of which groups belong to a specific client application and which items are in each group. When a client application creates a group on the server, it defines the group with properties that determine how frequently the server updates the client when item values change. For more information, see “Group properties” on [page 2.6](#).

The following illustration shows the hierarchy of components within an OPC server:



## Item properties

Item properties provide useful information about the data stored on a server. OPC servers support the following standard properties for each item:

### Item Data Type

The type of data stored by the server for this item. AutoMod allows models to read and write the following item data types:

| ID | Type    | Description  |
|----|---------|--|
| 2  | Integer | The item value is a whole number.  |
| 4  | Float   | The item value is a number containing a fractional part; float values are automatically cast to the Real data type when read in AutoMod. |
| 8  | String  | The item value is a sequence of characters (alphabetic, numeric, and/or symbol).   |
| 11 | Boolean | The item value is either 1 (True) or 0 (False); Boolean values are automatically cast to the Integer data type when read in AutoMod.     |

Many OPC servers report item types using a numerical ID number; the number that corresponds to each data type is shown in the ID column of the preceding table.

---

**Note** Other item data types, such as arrays, are not supported by AutoMod.

---

### Item Value

The value stored in the item. The type of value stored depends on the item's data type.

### Item Quality

The OPC Quality code assigned to this item by the server; the code provides information about the reliability of the item value. For a list of codes and their meanings, see Appendix A, "OPC Quality Codes."

## Item Timestamp

The date and time that the item value was written to the server. The format of the item timestamp is YYYY/MM/DD hh:mm:ss.

## Item Access Rights

An integer that indicates whether a client can read the item value, write the item value, or both read and write the item value. The item access rights are indicated as follows:

| Integer | Right      | Description  |
|---------|------------|--|
| 1       | Read-only  | You can read an item's value, but you cannot change it.  |
| 2       | Write-only | You can change an item's value, but not read it. Write-only items can represent a physical output with no readback capability. |
| 3       | Read/Write | You can both read and write the item's value.  |

## Server Scan Rate

The fastest rate, in milliseconds, that the server can obtain data from a data source. This rate represents a “best case” estimate by the server, which can be affected by system load and other factors.

AutoMod allows you to print the properties of all items on a particular server (see “Printing items on a server to the Messages dialog box” on [page 2.13](#)) or all items that are available through an open connection to a server (see “Printing connection information to the Messages dialog box” on [page 2.23](#)).

---

**Note** Additional properties might be maintained for each item, depending on the OPC server you are using. For more information about these properties, see the documentation that came with your server.

---

## Group properties

Because any number of OPC client applications can change item values on a server (for example, a control system and an AutoMod model can both write item values), the server allows clients to subscribe to specific items and receive automatic updates when the items' value changes. When a client subscribes to an item, the server scans the item's value at regular intervals to determine whether the value has changed. The properties of a subscribed item's group determines how frequently the server scans the item for changed values and the percentage by which a subscribed value must change before the server updates the subscribing client with the new value.

Each group that a client defines on an OPC server has two group properties:

### Update rate

Determines how frequently (in milliseconds) the server scans the value of subscribed items in the group to determine if the values have changed. The maximum rate at which the server can scan values for changes is limited by the server and consequently might be lower than the rate that you define for a group. For information about how frequently your OPC server can scan for updated item values, see the documentation that came with the OPC server.

### Deadband

Defines the percentage that a subscribed item's value must change, based on the item's range, before the server sends an updated value to the model. The deadband limits the number of updates that are sent to the model if small changes in subscribed items' values are not significant and do not require updating.

Both of these group properties are used to conserve resources on the server by limiting the server's communication with the model.

When a server scans a subscribed item and finds that its value has changed, the server determines whether or not to update the model with the new value by evaluating the following relation:

$$|(\text{last updated item value} - \text{current item value})| > \left( \frac{\text{Deadband}}{100} \right) \times (\text{EU High} - \text{EU Low})$$

The server only updates the model with a changed item value when the relation is true.

---

**Note** The relation uses engineering units (EU) high and low values to calculate the item's range. EU high and low values are defined when an item is created on an OPC server; for more information, see your server's documentation.

---

### Example - Updating a subscribed item value

This example demonstrates how a server determines whether to update a model with a subscribed item's value. Assume a subscribed item has an EU Low value of 15 and an EU High value of 25. Further, assume that the deadband defined for the item's group is 10 percent and the last updated item value was 18.5.

If the value on the server changes from 18.5 to 19, the server does *not* update the model with the new item value, because the relation is not true:

$$|(18.5 - 19)| > \left( \frac{10}{100} \right) \times (25 - 15) \text{ evaluates to } 0.5 > 1.0, \text{ which is false.}$$

However, if the item value increases again to 20, the server updates the model with the new item value, because the relation is true:

$| (18.5 - 20) | > \left( \frac{10}{100} \right) \times (25 - 15)$  evaluates to  $1.5 > 1.0$ , which is true.

---

## Emulation versus simulation

Designing an emulation model is very similar to designing a simulation model; however, there are some important differences:

- **An emulation model must include communication logic.** Emulation models open connections with one or more OPC servers, and read and write item values throughout a model run.
- **An emulation model is often more detailed than a simulation model.** Because the emulation model must provide the same responses to the OPC server as real system hardware, the model must be designed to respond to many system events that would otherwise not require custom processing during a simulation. (For example, an emulation model might be required to send signals to an OPC server when a load begins a pop-up transfer, when the transfer has completed lifting, when the load moves to the new section and when the transfer completes lowering.)
- **A simulation model contains control logic that can be omitted from an emulation model.** Whereas a simulation model directly controls devices and loads throughout a simulation, an emulation model often responds to signals from the control system, which controls system processes.

In many cases, a project will require both a simulation model for initial analysis and development, as well as an emulation model for testing a control system. If this is the case, you can use a copy of the simulation model as a starting point for developing the emulation model. To convert the simulation model into an emulation model, you will need to add logic for communicating with one or more OPC servers, and remove any redundant process control logic that is now handled by a control system.

## Designing a simulation model for use in emulation

If you know before building a simulation model that the model will be used for emulation, you can reduce the amount of time required to convert the model by planning for the conversion ahead of time. *As a general rule, design the simulation model using modular code that can easily be added or removed prior to emulation.*

For example, if part of the simulation logic will not be required during emulation (because it will be replaced by control system logic), you can write the logic in separate functions or procedures that can easily be removed. Similarly, you can add placeholders for future functions or procedures that will be necessary during emulation.

---

## Managing connections with OPC servers

An AutoMod model can communicate with an OPC server running on the local machine or on a remote machine on the local area network (LAN). You can open a connection between a model and one or more OPC servers by calling pre-defined connection functions in the AutoMod model. A function is also provided for verifying established connections.

Once opened, a connection with an OPC server remains active until you close it, either by closing the AutoMod runtime environment or by calling a function from within the model.

The following sections discuss these concepts in more detail.

### Opening connections

When you open a connection to an OPC server from an AutoMod model, you must indicate which items on the server you want to access from the model and the group to which each item belongs.

You can open a connection to one or more servers and create groups for the items you want to access in either of the following ways:

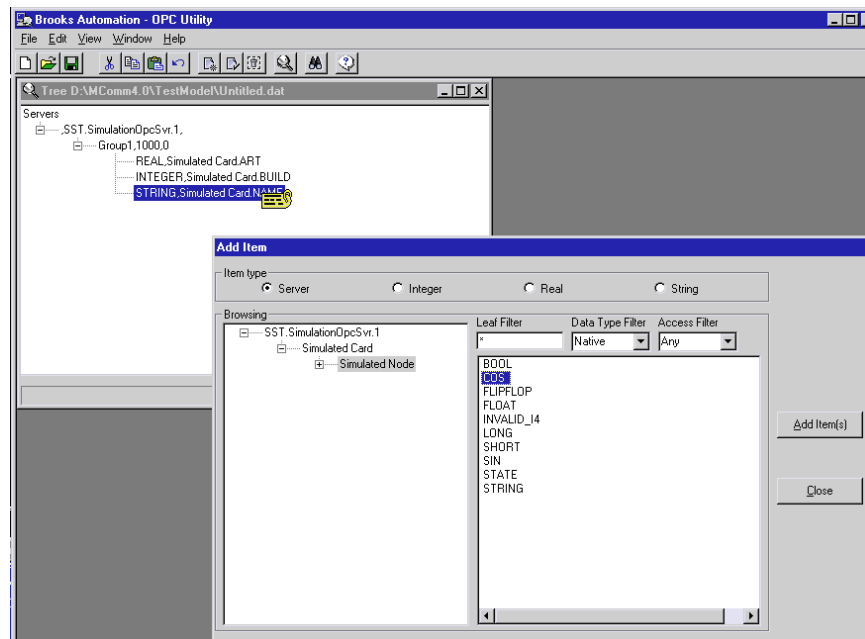
- By calling a function to read a configuration file created using the OPC Utility
- By calling separate functions to open a connection, create groups, and add items to each group

You can use these methods exclusively, or in combination, to set up connections with the appropriate OPC servers.

#### Opening connections using an OPC Utility configuration file

The easiest method of opening connections is to create a configuration file using the OPC Utility that is included with AutoMod. The utility allows you to connect to one or more OPC servers and list all available items on the server. Using the Tree dialog box in the utility, you can

then create a hierarchical configuration of OPC server components by adding groups to the connected servers and dragging items into each group as shown below:





After adding items to the configuration file, you can edit the items to link a variable in the AutoMod model with the item you are adding. Linked variables reduce the number of function calls required in the model by providing automatic updates to the model or OPC server when item values change. There are three types of links available:

| Link      | Description  |
|-----------|--|
| update    | <p>Linking a variable to an item on an OPC server using the <code>update</code> link causes the software to automatically update the value of the item on the server whenever the corresponding variable's value changes in the model. Use this link to automatically write variable values to the OPC server.</p> <p>To create an update link between a variable and item on an OPC server, you must also select <b>OPC Server Update</b> in the Model Defaults dialog box (for more information, see the “Using the Interface” chapter in the <i>AutoMod User's Guide</i>).</p>                |
| subscribe | <p>Linking a variable to an item on an OPC server using the <code>subscribe</code> link causes the software to subscribe to the item. Whenever the subscribed item's value on the server changes by a percentage that is greater than the deadband defined for the item's group, the model calls the <code>opccallback</code> function (see “Responding to changes in subscribed item values” on <a href="#">page 2.37</a>) and updates the value of the linked variable to match the value of the subscribed item. Use this link to automatically read variable values from the OPC server.</p> |
| set       | <p>Linking a variable to an item on an OPC server using the <code>set</code> link causes the software to update the value of the variable in the model only once—when the connection is opened. Use this link to initialize variables in the model with the value of items to which you do not want to subscribe</p> <p><b>Note</b> For more information about how to create a hierarchy of items and groups in a configuration file and for information on how to link items with AutoMod variables, see the OPC Utility online help.</p>   |

---

**Note** For more information about how to create a hierarchy of items and groups in a configuration file and for information on how to link items with AutoMod variables, see the OPC Utility online help.

---

After creating a configuration file, you can automatically open connections in the model by passing the configuration file name to the `OPCInit` function in the AutoMod model (for more information, see “`OPCInit` function” on [page B.8](#)). When called, the function reads the configuration file and automatically opens connections to each server listed in the file. The function creates the groups that are specified in the file and adds each of the listed items to the specified group so that

item values can be accessed in the model. If the configuration file specifies links between AutoMod variables and OPC items, the `OPCInit` function automatically creates the links.

---

**Note** The AutoMod model can only access items that have been added to a group; if a server contains items that were not added to a group (that is, the items are not listed in the configuration file), the model cannot read or write those item values during the model run.

---

## Example - Opening connections using a configuration file

In this example, the `OPCInit` function is called at the beginning of a model run (in the `model initialization` function) to open a connection to an OPC server. The function has one string parameter, which indicates the name and location of the configuration file. If the filename does not include a path, the software looks for the configuration file in the model directory.

```
begin model initialization function
  set V_result to OPCInit("config.dat")
  /* V_result is an integer variable; the function opens
     connections using the configuration file "config.dat." */
  if (V_result = -3) then begin
    print "Warning: the configuration file could not be
          found!" to message
    terminate /* stop the model */
  end
  return true /* return value ignored */
end
```

The integer variable `V_result` tests whether or not the connection file was found. The `OPCInit` function returns 0 if the configuration file is found and -3 if the file is not found (for more information, see “`OPCInit` function” on [page B.8](#)). If the function returns -3, a warning is printed and the `terminate` action ends the model run.

## Opening a connection using functions

AutoMod provides several functions that you can call in model logic to open a connection to an OPC server and set up the items and groups you want to access through the connection. These functions can be used in combination with, or instead of a configuration file (see “Opening connections using an OPC Utility configuration file” on [page 2.9](#)). The functions allow you to:

- Print all items on a server to the Messages dialog box.
- Open a connection to a server.
- Add groups to a server.
- Add items to a group.


**Printing items on a server to the Messages dialog box** To open a connection between a model and an OPC server using functions, you must first know the name of the server and the items on the server that you want the model to access. The `OPCListItems` function allows you to print all items on a specific server and all properties of each item to the Messages dialog box. If you do not know the name of the server to which you want to connect, you can print a list of all the OPC servers running on a specified machine, and all items for each server. For information about the properties that are printed for each item, see “Item properties” on [page 2.5](#).

The `OPCListItems` function requires two string parameters: the first parameter identifies the machine that is running the server(s) and the second parameter identifies the name of the server for which you want to list items. Passing a `null` value in the second parameter causes the function to list all servers and items on the specified machine. For more information, see “`OPCListItems` function” on [page B.10](#).

---

**Tip** The `OPCListItems` function is only meant for use during the model building process. After running the model once, you can open and print the `<modelname>.messages` file in the model directory, which contains all messages printed to the Messages dialog box during the simulation. After printing the list of servers and their items, you can refer to the printed list when opening connections and then comment out the `OPCListItems` function or remove it from your model.

---

**Important**  Some OPC servers do not support this feature; using the `OPCListItems` function with these servers might cause the servers to malfunction or might cause the model to become unresponsive. In this case, refer to the documentation that came with your server for information about how to list the items on the server.

**Example 1 - Printing items on all OPC servers on a specified machine** In this example, the `OPCListItems` function is used in the `model initialization` function to print a list of all the OPC servers and items on a networked machine, which is identified by its IP address.

```
begin model initialization function
  call OPCListItems("10.43.255.14", null)
  return true /* return value ignored */
end
```

The function requires two string parameters. The first parameter indicates the machine that is running the OPC server(s). The second parameter indicates the name of the OPC server for which you want to list the items; passing a `null` value in the second parameter, as shown

in this example, causes the function to list items for all servers running on the specified machine. For more information, see “OPCListItems function” on [page B.10](#)

---

**Note** AutoMod provides several ways to specify the machine on which the server is running (for more information, see “Defining a reference to a machine” on [page B.26](#)). Depending on how the server was installed, some methods might fail while others succeed. If you get errors opening a connection using a machine name, try using an IP address or one of the other methods.

---

Sample output for a machine that is running only one server, is shown below:

```
0.00:
0.00: Machine 10.43.255.14 has 1 servers:
0.00:     Server DLLTestSvr at 10.43.255.14 has 4 items:
0.00:         Item TempT
0.00:             Item Canonical Data Type: 4
0.00:             Item Value: 19.200001
0.00:             Item Quality: 192
0.00:             Item Timestamp: 9/17/01 7:33:30 AM
0.00:             Item Access Rights: 3
0.00:             Server Scan Rate: 10.000000
0.00:             EU Units:
0.00:             Item Description: Celsius temp
0.00:         Item LCount
0.00:             Item Canonical Data Type: 2
0.00:             Item Value: 53
0.00:             Item Quality: 192
0.00:             Item Timestamp: 9/17/01 7:33:30 AM
0.00:             Item Access Rights: 3
0.00:             Server Scan Rate: 10.000000
0.00:             EU Units:
0.00:             Item Description: Load count
0.00:         Item RStatus
0.00:             Item Canonical Data Type: 8
0.00:             Item Value: Proc
0.00:             Item Quality: 192
0.00:             Item Timestamp: 9/17/01 7:33:30 AM
0.00:             Item Access Rights: 3
0.00:             Server Scan Rate: 10.000000
0.00:             EU Units:
0.00:             Item Description: Resource status
0.00:         Item Press
0.00:             Item Canonical Data Type: 4
0.00:             Item Value: 15.530000
0.00:             Item Quality: 192
0.00:             Item Timestamp: 9/17/01 7:33:30 AM
0.00:             Item Access Rights: 3
```

```
0.00:          Server Scan Rate: 10.000000
0.00:          EU Units:
0.00:          Item Description: Pressure
0.00:
```

---

This software prints this list of servers and items to the Messages dialog box.

**Example 2 - Printing items on a specific OPC server** In this example, the `model initialization` function calls the `OPCListItems` function to list all items on the server `DLLTestSvr`, which is running on a networked machine.

```
begin model initialization function
  call OPCListItems("10.43.255.14", "DLLTestSvr")
  return true /* return value ignored */
end
```

**Example 3 - Printing items on all OPC servers running on the local machine**

In this example, the `model initialization` function calls the `OPCListItems` function to list all items on all OPC servers that are running on the local machine (the same machine as the AutoMod model).

```
begin model initialization function
  call OPCListItems(null, null)
  return true /* return value ignored */
end
```

**Opening the connection** To open a connection to an OPC server using functions, call the `OPCConnectServer` function at the beginning of the model run. The function allows you to connect to a server running on the same machine as the AutoMod model (the local machine) or to a server on a machine that can be accessed through a network (a remote machine).

The `OPCConnectServer` function requires three string parameters: the first parameter indicates the machine on which the server is running, the second parameter indicates the name of the OPC server to which you want to connect, and the third parameter assigns an alias to the server that you can use in other functions. Assigning an alias to a server is useful if you are connecting to multiple OPC servers that

have the same name but are running on different machines and is also useful for brevity when writing model logic. For more information, see “OPCConnectServer function” on [page B.5](#)

---

**Note** There are many ways to specify the machine on which the server is running (for more information, see “Defining a reference to a machine” on [page B.26](#)). Depending on how the server was installed, some methods might fail while others succeed. If you get errors opening a connection using a machine name, try using an IP address or one of the other methods.

**Tip** If you do not know the name of the server to which you want to connect, you can list the names of all servers and items on a specified machine by calling the `OPCListItems` function (see “Printing items on a server to the Messages dialog box” on [page 2.13](#)).

---

**Important** When opening connections to servers at your facility, ensure that you connect only to servers that are dedicated to testing. Connecting and writing values to a production server could result in system hardware damage or operator injury.

**Adding groups to a server** After opening a connection, you can call the `OPCAddGroup` function to define one or more groups on the server. Groups are used to organize items on the server; groups also limit how frequently the server scans subscribed items and updates the model with changed item values. For more information, see “What is an OPC server?” on [page 2.3](#).

---

**Note** For a server to automatically update a model with changed item values, you must subscribe to the group or to individual items within the group. For more information, see “Subscribing to groups and items” on [page 2.33](#).

---

**Important** Although AutoMod allows you to define a deadband for groups you create on the server, not all OPC servers support this feature. To use a group’s deadband to limit the amount of updates that are sent to a model, the following must be true:

- The OPC server must support deadband values.
- All items that are added to the group must be defined on the server with an Analog `dwEUInt`. (For information about how to define items with an Analog `dwEUInt`, see the documentation that came with your OPC server.)

If these conditions are not met, you must define a group with a deadband of zero, which causes the server to update the model when there is any change in a subscribed item’s value.

The `OPCAddGroup` function requires four parameters: the first parameter indicates the server to which you are adding a group, the second parameter defines the group name, the third parameter defines the group's update rate, and the fourth parameter defines the group's deadband. For more information about a group's update rate and deadband, see "Group properties" on [page 2.6](#). For more information about the `OPCAddGroup` function and its required parameters, see "OPCAddGroup function" on [page B.2](#)

**Subscribing to items when adding groups to a server** When you add a group to an OPC server using the `OPCAddGroup` function, you can simultaneously subscribe to all items in the group. Subscription is useful for automatically reading item values from the OPC server. Whenever the value of a subscribed item on the server changes by a percentage that is greater than the deadband defined for the item's group, the model calls the `opccallback` function (see "Responding to changes in subscribed item values" on [page 2.37](#)). The `opccallback` function allows the model to respond to the changed item value.

---

**Note** When you subscribe to all items in a group using the `OPCAddGroup` function, you cannot link the subscribed items to AutoMod variables. If you want AutoMod to automatically update the value of a linked variable when the value of a subscribed item changes, you must subscribe to the item either when adding the item to a group on the server (see "Adding items to a group" on [page 2.17](#)) or by calling the `OPCSubscribe` function to subscribe to the item during the simulation (see "Subscribing to groups and items" on [page 2.33](#)).

---

To automatically subscribe to all items in a group, append the keyword `subscribe`, enclosed in brackets, to the group name in the second parameter of the `OPCAddGroup` function. For example:

```
call OPCAddGroup("TestSvr", "GRFirst [subscribe]", 1, 10)
```

In this example, the software creates the group `GRFirst` on the server `TestSvr`. The group is defined with an update rate of 1 millisecond and a deadband of 10 percent. The software automatically subscribes to all items in the group.

**Adding items to a group** After creating one or more groups on the server, you can call the `OPCAddItem` function to add items to the created groups. Adding an item allows the model to access the value for that item on the server. You can then read and write the item's value during a model run (see "Reading item values" on [page 2.30](#) and "Writing item values" on [page 2.25](#)).

The `OPCAddItem` function requires four string parameters: the first parameter indicates the server containing the item, the second parameter indicates the group to which you are adding the item, the third parameter indicates the item name, and the fourth parameter indicates the item's type.

---

**Note** If you add an item to a group that does not yet exist on the server, the `OPCAddItem` function attempts to create the group with an update rate of 1000 milliseconds and a deadband of zero (for more information, see “Group properties” on [page 2.6](#)).

---

**Creating items on an OPC server** If you add an item that does not yet exist on the server to a group using the `OPCAddItem` function, the function automatically attempts to create the item on the server.

---

**Note** Some OPC servers do not allow AutoMod to create items (for information on how to create an item on your OPC server, refer to the documentation that came with your server.)

---

If you create an item on a server using the `OPCAddItem` function, you can initialize the item's value on the server. To initialize the item's value, append the keyword `initialize` followed by the item value, both enclosed in brackets, to the item name in the third parameter of the `OPCAddItem` function. For example:

```
call OPCAddItem("TestSvr", "GRFirst", "NewIntItem[initialize 0]",  
               "Integer")
```

In this example, the `OPCAddItem` function creates the integer item `NewIntItem` in the group `GRFirst` on the server `TestSvr`. The item is initialized with a value of 0.

---

**Note** Regardless of whether you are using the `OPCAddItem` function to create a new item on the server or add an existing item, you must call the `OPCAddItem` function for each item that you want to read from or write to during the model run. If a server contains items that were not added to a group by AutoMod, the model cannot read or write those item values during the model run.

---



**Linking added items with AutoMod variables** When you add an item to a group, you can also link the item to a variable in the AutoMod model. Linked variables reduce the number of function calls required in the model by providing automatic updates to the model or OPC server when item values change. There are three types of links available:

| Link      | Description  |
|-----------|--|
| update    | <p>Linking a variable to an item on an OPC server using the <code>update</code> link causes the software to automatically update the value of the item on the server whenever the corresponding variable's value changes in the model. Use this link to automatically write variable values to the OPC server.</p> <p><b>Note</b> To create an update link between a variable and item on an OPC server, you must also select <b>OPC Server Update</b> in the Model Defaults dialog box (for more information, see the “Using the Interface” chapter in the <i>AutoMod User's Guide</i>).</p>  |
| subscribe | <p>Linking a variable to an item on an OPC server using the <code>subscribe</code> link causes the software to subscribe to the item. Whenever the subscribed item's value on the server changes by a percentage that is greater than the deadband defined for the item's group, the model calls the <code>opccallback</code> function (see “Responding to changes in subscribed item values” on <a href="#">page 2.37</a>) and updates the value of the linked variable to match the value of the subscribed item. Use this link to automatically read variable values from the OPC server</p> <p><b>Note</b> This type of link can also be created by calling functions in the model (see “Subscribing to groups and items” on <a href="#">page 2.33</a>).</p> |
| set       | <p>Linking a variable to an item on an OPC server using the <code>set</code> link causes the software to update the value of the variable in the model only once—at the time the <code>OPCAddItem</code> function is called. Use this link to initialize variables in the model with the value of items to which you do not want to subscribe.</p>   |

To link an item on an OPC server with a variable in an AutoMod model, append a link type and variable name, enclosed in brackets, to the item name in the third parameter of the `OPCAddItem` function. Examples are shown below:

```
call OPCAddItem("TestSvr", "GR1Sens", "WtrP[subscribe V_WtrP]", "Real")
call OPCAddItem("TestSvr", "GR1Mot", "Mot1[update V_Mve11]", "Real")
call OPCAddItem("RsLinx", "GRInit", "Ln[set V_Ln]", "Integer")
```

The first function call subscribes to the item `WtrP` and links the item's value to the variable `V_Wtrp`; whenever the item's value changes on the server, the new value is written to the variable in the model. The second function call links the item `Mot1` to the variable `V_Mve11`; whenever the variable's value changes in the model, the new value is

written to the item on the OPC server. The last function call links the item `Ln` with the variable `V_Ln`; the value of the variable is initialized to the item's value on the server.

For more information about the `OPCAddItem` function and its required parameters, see “[OPCAddItem function](#)” on [page B.3](#)

**Example 1 - Opening a connection using functions** In this example, the model opens a connection to the server `DLLTestSvr` running on the networked machine `TestMach3`. The connection is opened in the `model` initialization function at the beginning of each model run.

After opening a connection, the model creates two groups on the server and adds three items to the first group and one to the second. When adding groups and items, the return value of each function is passed to the user-defined function `F_isError`, which checks the function's return value and stops the model if an error occurs.

```
begin model initialization function
  set Vresult to OPCConnectServer("TestMach3", "DLLTestSvr",
    "TestSvr")
  /* The first parameter indicates the name of the remote
    computer; the second parameter indicates the name of
    the OPC server; the third parameter assigns an alias
    to the server. */
  if (Vresult < 0) then
    begin
      print "Error: OPCConnectServer function returned "
        Vresult ". Connection not established." to message
        terminate
    end
  call F_isError(OPCAddGroup("TestSvr", "GR1Sens", 1000, 0),
    "GR1Sens")
  /* The first parameter of F_isError is passed the return
    value of the OPCAddGroup function; the second
    parameter identifies the group being added. */
  call F_isError(OPCAddGroup("TestSvr", "GR1Fluct", 2000, 10),
    "GR1Fluct")
  call F_isError(OPCAddItem("TestSvr", "GR1Sens", "Temprt",
    "Real"), "Temprt")
  /* The first parameter of F_isError is passed the return
    value of the OPCAddItem function; the second parameter
    identifies the item being added. */
  call F_isError(OPCAddItem("TestSvr", "GR1Sens",
    "LCount[update V_Lcount]", "Integer"), "LCount")
  /* The added item is linked with the variable V_Lcount.*/
  call F_isError(OPCAddItem("TestSvr", "GR1Sens", "RStatus",
    "String"), "RStatus")
  call F_isError(OPCAddItem("TestSvr", "GR1Fluct", "Press",
    "Real"), "Press")
  return true /* return value ignored */
end
```

The first group added to the server, `GR1Sens`, is scanned for changes every second and subscribed items are updated whenever an item value changes. The second group, `GR1Fluct`, scans for updates every two seconds and requires that subscribed item values change by at least 10 percent before an update is sent to the model.

The item `LCount` is linked with the variable `V_Lcount` in the model. The link type is `update`, so whenever the variable's value changes in the model, the value is automatically written to the item on the OPC server.

The `OPCAddGroup` and `OPCAddItem` functions each return a negative value if an error occurs. The function `F_isError` prints an error message and stops the model run if any function return value is less than zero:

```
begin F_isError function
  if (PrmRetVal < 0) then begin
    print "Error adding item or group: " PrmAddName
      to message
    print "Request returned error: " PrmRetVal to message
    terminate
  end
  return false /* return value ignored */
end
```

**Example 2 - Opening a connection to RSLinx using functions** RSLinx is a Rockwell communications server that communicates between plant devices and software applications. This example demonstrates how to open a connection to an RSLinx server that is running on the same machine as the AutoMod model.

---

**Note** When referencing items on the server, the name of the item's private group in RSLinx is enclosed in brackets and precedes the item name, as shown in the `OPCAddItem` functions below. The private group name must also be included in brackets when calling functions to read or write item values (see "Reading item values" on [page 2.30](#) and "Writing item values" on [page 2.25](#)).

---

```
begin model initialization function
  set Vresult to OPCConnectServer(null, "RSLinx OPC Server",
    "RSLinx")
  /* The first parameter's null value indicates the OPC
    server is running on the local machine; the second
    parameter indicates the name of the OPC server; the
    third parameter assigns an alias to the server. */
  if (Vresult < 0) then
    begin
      print "Error: OPCConnectServer function returned "
        Vresult ". Connection not established." to message
      terminate
    end
    call F_isError(OPCAddGroup("RSLinx", "GR1Sens", 1000, 0),
      "GR1Sens")
  /* The first parameter of F_isError is the return value
```

```

        of the OPCAddGroup function; the second parameter
        identifies the group being added. */
    call F_isError(OPCAddGroup("RSLinx", "GR1Fluct", 2000, 10),
        "GR1Fluct")
    call F_isError(OPCAddItem("RSLinx", "GR1Sens",
        "[GrpTest]Temprt", "Real"), "Temprt")
    /* The added item "Temprt" belongs to the private group
        "GrpTest" on the RSLinx server. */
    call F_isError(OPCAddItem("RSLinx", "GR1Sens",
        "[GrpTest]LCount[update V_Lcount]", "Integer"),
        "LCount")
    /* The added item is linked with the variable V_Lcount.*/
    call F_isError(OPCAddItem("RSLinx", "GR1Sens",
        "[GrpTest]RStatus", "String"), "RStatus")
    call F_isError(OPCAddItem("RSLinx", "GR1Fluct",
        "[GrpTest]Press", "Real"), "Press")
    return true /* return value ignored */
end

```

The item `LCount` is linked with the variable `V_Lcount` in the model. The link type is `update`, so whenever the variable's value changes in the model, the value is automatically written to the item on the OPC server. The user-defined function `F_isError` is defined the same as in the previous example (see "Example 1 - Opening a connection using functions" on [page 2.20](#)).

## Verifying connections

After opening connections, creating groups, and adding items to each group, there are two ways to verify the open connections from within a model:

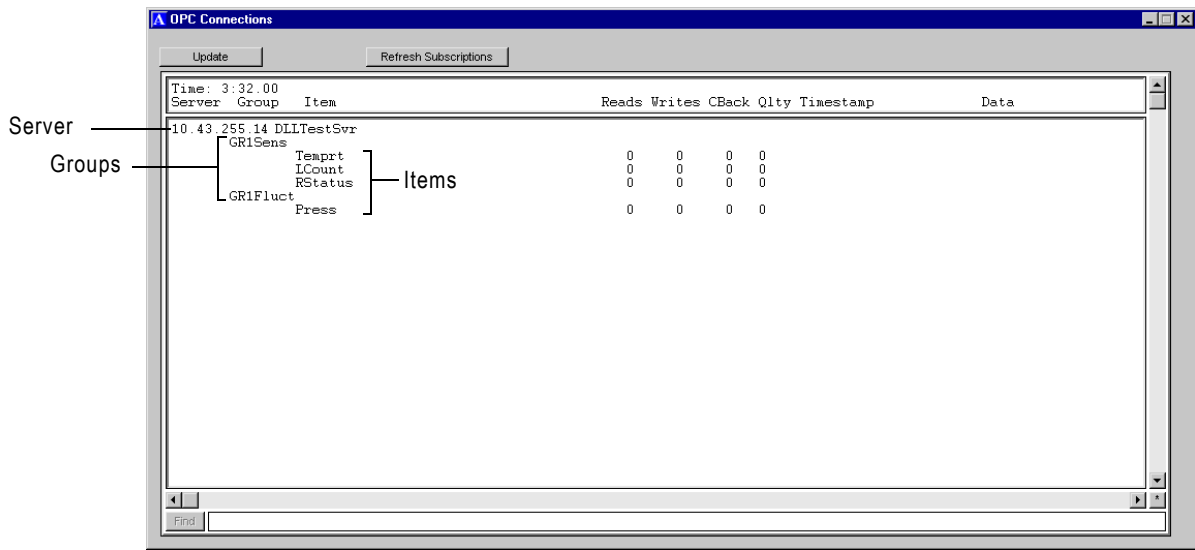
- Open a dialog box that displays server, group, and item information during a model run.
- Print server, group, and item information to the Messages dialog box during a model run.

### Displaying connection information during a model run

► **To display a list of the currently connected servers, the groups that have been defined, and the items that have been added to each group**

1. Run the model.
2. In the Simulation window, select **OPC Connections** from the **Loads/Variables** menu.

The OPC Connections dialog box opens:



The dialog box lists each server that is currently connected to the model, each group that the model has created on each server, and the items in each group.

**Note** The OPC Connections dialog box contains buttons that update the display of current item values (see “Displaying item values and update statistics” on [page 2.46](#)) and refresh subscriptions to items on the server (see “Dynamically refreshing item subscriptions during a model run” on [page 2.36](#)). You do not need to use these options when verifying connections.

## Printing connection information to the Messages dialog box

To verify connections by printing the connection information to the Messages dialog box, call the `OPCListConnections` function in the model. The function lists the currently connected servers, the groups that have been defined, and the items that have been added to each group. For information about the properties printed for each item, see “Item properties” on [page 2.5](#).

**Note** Some OPC servers do not support this feature; if the function does not list any connections, refer to the documentation that came with your server to determine how to list open connections to the server.

For more information, see “`OPCListConnections` function” on [page B.9](#)


**Example - Printing connection information to the Messages dialog box** In this example, the `OPCListConnections` function is called in the `model ready` function at the beginning of a simulation:

```
begin model ready function
    set V_NumItems to OPCListConnections()
    /* The function's return value is the number of items added
       to the connected OPC servers. */
    print "Number of added items: " V_NumItems to message
    return true
end
```

The following example shows the printed output for a simple configuration consisting of one server connection:

```
0.00: Machine 10.43.255.14 Server DLLTestSvr Alias TestSvr
0.00:   Group GRISens Req Rate 1000.000000 Actual Rate 1000.000000 Deadband: 0.000000
0.00:     Real Item TempT
0.00:       Item Canonical Data Type: 4
0.00:       Item Value: 19.200001
0.00:       Item Quality: 192
0.00:       Item Timestamp: 9/17/01 7:33:30 AM
0.00:       Item Access Rights: 3
0.00:       Server Scan Rate: 10.000000
0.00:       EU Units:
0.00:       Item Description: Celsius temp
0.00:     Int Item LCount
0.00:       Item Canonical Data Type: 2
0.00:       Item Value: 53
0.00:       Item Quality: 192
0.00:       Item Timestamp: 9/17/01 7:33:30 AM
0.00:       Item Access Rights: 3
0.00:       Server Scan Rate: 10.000000
0.00:       EU Units:
0.00:       Item Description: Load count
0.00:     String Item RStatus
0.00:       Item Canonical Data Type: 8
0.00:       Item Value: Proc
0.00:       Item Quality: 192
0.00:       Item Timestamp: 9/17/01 7:33:30 AM
0.00:       Item Access Rights: 3
0.00:       Server Scan Rate: 10.000000
0.00:       EU Units:
0.00:       Item Description: Resource status
0.00:   Group GRIFluct Req Rate 2000.000000 Actual Rate 2000.000000 Deadband: 10.000000
0.00:     Real Item Press
0.00:       Item Canonical Data Type: 4
0.00:       Item Value: 15.530000
0.00:       Item Quality: 192
0.00:       Item Timestamp: 9/17/01 7:33:30 AM
0.00:       Item Access Rights: 3
0.00:       Server Scan Rate: 10.000000
0.00:       EU Units:
0.00:       Item Description: Pressure
0.00: Number of added items: 4
```

## Closing connections

It is important that open connections to an OPC server be closed when they are no longer needed. Closing connections to an OPC server frees the server resources that are used to maintain the connection. All connections to OPC servers close automatically at the end of a model run and when the AutoMod runtime environment is closed (either because you select **Exit** from the **File** menu, or because you click **Close**  in the Simulation window).

If you want to manually close a connection to one or more servers during a model run, you can call the `OPCDisconnect` function. The `OPCDisconnect` function requires one string parameter: the name or alias of the server from which you want to disconnect. You can disconnect from all servers by passing a `null` value to the function. For more information, see “`OPCDisconnect` function” on [page B.6](#)

### Example 1 - Disconnecting from all OPC servers

To disconnect from all OPC servers, pass a `null` value to the `OPCDisconnect` function as shown below:

```
call OPCDisconnect(null) /* close all connections */
```

### Example 2 - Disconnecting from a specific OPC server

To disconnect from a specific OPC server, “pass” the server name or alias to the `OPCDisconnect` function as shown below:

```
call OPCDisconnect("TestSvr")
/* closes connection to TestSvr only */
```

---

## Writing item values

Models can write to an OPC server in the same way that devices write to the server to indicate the status of the device, the device’s configuration parameters, and so on. When writing to an OPC server, you can perform a synchronous write with or without validation.

A *synchronous write with validation* sends an item’s value to an OPC server and then delays the model run until either the updated value is verified on the server or until the write times out. After performing a synchronous write, the model verifies that the item value has been written to the server by reading the current item value from the server and comparing it to the value sent from the model. Use a write with validation to ensure that the OPC server updates the item value before allowing the model to continue processing.

A *synchronous write without validation* sends an item’s value to an OPC server and continues processing, without waiting to verify that the server has updated the item value. This type of write is useful if

you are writing a large number of item values to a server at the same time, and want to delay verification until all values have been sent by the model.

**Important** For a model to write to an item’s value, the item must be defined on the OPC server with writable access rights. You can print the access rights of all items the model has added to groups on the server (see “Printing connection information to the Messages dialog box” on [page 2.23](#)). If an item value is not writable, any attempt by a model to write the value is ignored. For information about changing item access rights, see the documentation that came with your OPC server.

## Writing item values synchronously with validation

To write an item’s value to an OPC server synchronously with validation, call a function in the AutoMod model; the function you use depends on the value’s type:

| To write an item value of this type: | Call this function:                             |
|--------------------------------------|---|
| Float                                | OPCWriteRealWait(<Item>, <Value>, <Timeout>)    |
| Integer or Boolean                   | OPCWriteIntegerWait(<Item>, <Value>, <Timeout>) |
| String                               | OPCWriteStringWait(<Item>, <Value>, <Timeout>)  |

Each of the functions requires three parameters: the first parameter indicates the item on the server to which you want to write a value (for more information, see “Defining a reference to an item” on [page B.26](#)), the second parameter indicates the new item value, and the third parameter indicates how long in seconds the model should wait to verify that the item is written to the server before continuing processing.

**Note** When referring to items on the server, you might need to include a server name or alias in the item reference, depending on whether items on different OPC servers share the same name. If an item name is unique, you can improve model performance by referencing the item name alone. For more information, see “Defining a reference to an item” on [page B.26](#).

Because the model verifies the write by comparing the current item value on the server with the value sent from the model, any changes that the server makes to received values can cause the write to the server to timeout. For example, if you are writing a string value and the OPC server truncates the string, the compared values will not match and the write will timeout. Similarly, if the OPC server rounds or truncates a real value, resulting in a change in the value that exceeds the value’s tolerance, the write will timeout. Each real value’s



tolerance is determined by the deadband defined for the item's group. If the following relation is true, the model assumes the item value has been written to the server and does *not* timeout:

$$\frac{|\text{value read from server} - \text{value written to server}|}{\text{value read from server}} < \frac{\text{deadband}}{100}$$

For more information about defining a group's deadband, see "Adding groups to a server" on [page 2.16](#).

### Example - Writing item values synchronously with validation

In this example, an arrayed process (P\_downtime) simulates random downtimes for three machines. Each time the process' arriving procedure takes down a machine, it writes to a Boolean item on the OPC server to indicate the machine status (1 indicates the machine is up, and 0 indicates the machine is down for repairs).

In this example, an arrayed process (P\_downtime) simulates random downtimes for three machines. Each time the process' arriving procedure takes down a machine, it synchronously writes to a Boolean item on the OPC server to indicate the machine status (1 indicates the machine is up, and 0 indicates the machine is down for repairs). The model delays up to one second to verify that the item value was written to the server before timing out and calling the user-defined F\_WriteFailed function.

```
begin P_downtime arriving procedure
  print "StatMach" procindex to LA_Machname
  /* LA_Machname is a load attribute of type String */
  while 1 = 1 do
    begin
      wait for e 200 min /* up time */
      take down R_Mach(procindex)
      set V_WriteResult to OPCWriteIntegerWait(LA_Machname,
        0, 1.0)
      /* update machine status on the OPC server */
      if V_WriteResult < 0 then call F_WriteFailed(LA_Machname)

      wait for e 10 min /* down time */

      bring up R_Mach(procindex) /* repairs finished */
      set V_WriteResult to OPCWriteIntegerWait(LA_Machname,
        1, 1.0)
      /* update machine status on the OPC server */
      if V_WriteResult < 0 then call F_WriteFailed(LA_Machname)
    end
  end
end
```

At the beginning of the model run, three initialization loads are sent to the P\_downtime process, where they remain throughout the simulation. Loads in the procedure first print the item name (StatMach1, StatMach2, or StatMach3) to a load attribute. Loads use the attribute in the OPCWriteIntegerWait function, which they call whenever they take down or bring up a machine.

Each time a value is written to the server, the procedure checks the return value of the writing function. If an error occurs or if the function times out (the return value is less than zero), the user-defined function `F_WriteFailed` (not shown) is called to print an error message and terminate the model run.

## Writing item values synchronously without validation

To write item values to an OPC server synchronously without validation, call a function in the AutoMod model to write each item value, then call the `OPCFlushWrite` function to verify that all the values were successfully written to the server. The function you use to write each item's value depends on the value's type:

**To write an item value of this type: Call this function:**

|                    |  |
|--------------------|--|
| Float              | <code>OPCWriteReal(&lt;Item&gt;,&lt;Value&gt;)</code>    |
| Integer or Boolean | <code>OPCWriteInteger(&lt;Item&gt;,&lt;Value&gt;)</code> |
| String             | <code>OPCWriteString(&lt;Item&gt;,&lt;Value&gt;))</code> |

Each of the functions requires two parameters: the first parameter indicates the item on the server to which you want to write a value (for more information, see “Defining a reference to an item” on [page B.26](#)), and the second parameter indicates the new item value.

Because writing to a server without validation does not verify that the item values were successfully written to the server, you can call the `OPCFlushWrite` function to verify each written value (see “`OPCFlushWrite` function” on [page B.7](#) for more information). The `OPCFlushWrite` function requires one parameter that indicates the item on the server that you want to verify. If you pass a `null` value to this parameter, the function verifies all values that have been written to the server but have not yet been verified in the model.

**Note** When referring to items on the server, you might need to include a server name or alias in the item reference, depending on whether items on different OPC servers share the same name. If an item name is unique, you can improve model performance by referencing the item name alone. For more information, see “Defining a reference to an item” on [page B.26](#).

### Example - Writing item values synchronously without validation

In this example, a function writes several statistics to items on the OPC server and then calls the `OPCFlushWrite` function to verify that the values were written successfully.

```

begin F_WriteStats function
  set V_count to 1
  while V_count < 5 do begin
    print "AGV(" V_count ")" to V_ItemBaseName
    /* V_ItemBaseName is a variable of type String */
    set V_VehPtr to pm:AGV(V_count)
    /* V_VehPtr is a variable of type VehiclePtr */

    print "Acc_" V_ItemBaseName to V_ItemName
    set V_IVReal to V_VehPtr acceleration
    call OPCWriteReal(V_ItemBaseName,V_IVReal)

    print "Dec_" V_ItemBaseName to V_ItemName
    set V_IVReal to V_VehPtr deceleration
    call OPCWriteReal(V_ItemBaseName,V_IVReal)

    print "Vel_" V_ItemBaseName to V_ItemName
    set V_IVReal to V_VehPtr velocity
    call OPCWriteReal(V_ItemBaseName,V_IVReal)

    print "Loc_" V_ItemBaseName to V_ItemName
    print V_VehPtr current location to V_IVString
    call OPCWriteString(V_ItemBaseName,V_IVString)

    print "TP_" V_ItemBaseName to V_ItemName
    set V_IVInt to V_VehPtr trips going to park
    call OPCWriteInteger(V_ItemName,V_IVInt)

    print "TR_" V_ItemBaseName to V_ItemName
    set V_IVInt to V_VehPtr trips retrieving
    call OPCWriteInteger(V_ItemName,V_IVInt)

    print "TD_" V_ItemBaseName to V_ItemName
    set V_IVInt to V_VehPtr trips delivering
    call OPCWriteInteger(V_ItemName,V_IVInt)
    increment V_count by 1
  end

  set V_Result to OPCFlushwrite(null)
  /* variable of type Integer */
  if V_Result < 0 then begin
    print "ERROR: F_WriteStats could not write all items;
      terminating"to message
    terminate
    return false
  end
  else return true
end

```

---

## Reading item values

Throughout a model run, a model can read item values from the OPC server in the same way that devices read item values to receive instructions from the control system.

There are two ways to read item values from an OPC server:

- You can call a function to read an item's current value at a specific time during the model run.
- You can subscribe to an item so the OPC server will continuously update the model with changes in the item's value throughout the model run. When the value of a subscribed item changes by a percentage that is greater than the deadband defined for the item's group, the OPC server automatically updates the value of an AutoMod variable that corresponds to the item and calls the `opccallback` function defined for the item in the model.

Each time an item value is read from the OPC server (either because the model calls a function to read the value or because the OPC server updates the value of a subscribed item), the model automatically updates two system-defined AutoMod variables:

| Variable     | Description   |
|--------------|---|
| OPCQuality   | <p>An integer that indicates the reliability of the value that was read from the server. For more information about OPC quality codes and their meanings, see “OPC Quality Codes” on <a href="#">page A.1</a>.</p> <p><b>Note</b> If you attempt to read the value of an item that does not exist on the specified server, the OPCQuality value is set to -1.</p>   |
| OPCTimestamp | <p>The date and time that the item value was written to the server. The format of the item timestamp is YYYY/MM/DD hh:mm:ss.</p> <p>You can use these variables in any procedure, function, or subroutine in the model logic.</p> <p><b>Important</b> For a model to read an item's value, the item must be defined on the OPC server with readable access rights. You can print the access rights of all items the model has added to groups on the server (see “Printing connection information to the Messages dialog box” on <a href="#">page 2.23</a>). For information about changing item access rights, see the documentation that came with your OPC server.</p> |

## Reading an item's value at a specific time during a model run

You can find a single item's value at any point during a model run by calling a function to read the value from the server. At the beginning of a simulation, this method of reading item values is useful for initializing values that do not change throughout the run. The function you use to read item values depends on the type of value you are reading:

**To read an item value of this type: Call this function:**

|                    |   |
|--------------------|---|
| Float              | <code>OPCReadReal(&lt;Item&gt;)</code>    |
| Integer or Boolean | <code>OPCReadInteger(&lt;Item&gt;)</code> |
| String             | <code>OPCReadString(&lt;Item&gt;)</code>  |

Each function requires a string parameter that indicates the item from which you want to read a value.

**Note** When referring to items on the server, you might need to include a server name or alias in the item reference, depending on whether items on different OPC servers share the same name. If an item name is unique, you can improve model performance by referencing the item name alone. For more information, see “Defining a reference to an item” on [page B.26](#).

You can also automatically cast data read from the server to a different type, depending on which function you use to read the data. For example, if you call the `OPCReadInteger` function to read a Boolean or float value, the software automatically casts the value to an integer in the model.

Each time an item value is updated by the server, the AutoMod variables `OPCQuality` and `OPCTimestamp` are automatically updated with information about the value. You can use these variables to check the quality of the data or the time it was received from the server (for more information, see “Reading item values” on [page 2.30](#)).

### Example - Reading item values

In this example, an initialization procedure (`P_Init`) reads float values from the server `TestSvr` and uses them to set the velocity of four conveyor sections. Although initialized from the server, these values do not change throughout the model run, so the model does not subscribe to the velocities on the server.

After reading each item value, the procedure calls the user-defined function `F_isReadError` to check the value's quality. If the quality is not “good,” the function prints a warning to the Messages dialog box and the `P_Init` procedure ignores the item value.

```

begin P_Init arriving procedure
  set V_velocity to OPCReadReal("TestSvr,Sec1Vel")
  /* V_velocity is a variable of type velocity */
  /* item reference uses the server alias and item name */
  if (F_isReadError("Sec1Vel") is false) then
    set conv:sec1 velocity to V_velocity

  set V_velocity to OPCReadReal("Sec2Vel")
  /* item reference uses the item name alone */
  if (F_isReadError("Sec2Vel") is false) then
    set conv:sec2 velocity to V_velocity

  set V_velocity to OPCReadReal("Sec3Vel")
  if (F_isReadError("Sec3Vel") is false) then
    set conv:sec3 velocity to V_velocity

  set V_velocity to OPCReadReal("Sec4Vel")
  if (F_isReadError("Sec4Vel") is false) then
    set conv:sec4 velocity to V_velocity

  send to die
end

```

---

**Note** If you include a server name or alias in an item reference, the server and item names must be separated by a comma and cannot include any spaces, as shown.

---

After setting the velocity of each conveyor section, the load that executes the initialization procedure is sent to die.

The user-defined function that checks the quality of read values is defined as follows:

```

begin F_isReadError function
  if (OPCQuality < 192) then
    begin
      print "Warning: OPC Quality is not good. Attempt to
        read a value for item " PrmItemName " returned
        quality code " OPCQuality to message
      return true
    end
  else return false
end

```

For more information about quality codes, see Appendix A, "OPC Quality Codes."

## Subscribing to groups and items

Item subscription provides a way for a model to automatically respond to changes in item values on an OPC server. There are several ways to subscribe to items on an OPC server:

- Subscribe to all items in a group when defining the group on the server (see “Subscribing to items when adding groups to a server” on [page 2.17](#)).
- Subscribe to an item when adding it to a group (see “Linking added items with AutoMod variables” on [page 2.19](#)).
- Subscribe to one or more items when opening connections using a configuration file (see “Opening connections using an OPC Utility configuration file” on [page 2.9](#)).
- Subscribe to one or more items during a simulation (discussed below).

To subscribe to items on an OPC server during a simulation, call the `OPCSubscribe` function. The `OPCSubscribe` function allows you to subscribe to a group of items or a specific item on the server. You can call the function as many times as needed to subscribe to the appropriate items on the server. When you subscribe to an individual item, you can associate it with an AutoMod variable, which is automatically set to the item’s value.

The `OPCSubscribe` function requires two string parameters: the first parameter identifies the item or group to which you want to subscribe. The second parameter identifies the variable that you want to associate with the subscribed item (the second parameter must be `null` when subscribing to a group). For more information, see “`OPCSubscribe` function” on [page B.14](#)

OPC servers repeatedly scan item values for changes. (You define the rate at which a server scans items when you create a group for the items on the server. For more information, see “Adding groups to a server” on [page 2.16](#).) While scanning items, a server updates the model if a subscribed item value changes by a percentage that is greater than the deadband value defined for the item’s group (for more information, see “Group properties” on [page 2.6](#)).

When a server updates the model with a new item value, the model checks to see if the item is associated with a corresponding AutoMod variable; if so, the model automatically updates the variable’s value to match the subscribed item’s value. Each time the server updates the model with a new item value, the model also automatically calls the `opccallback` function defined for that item. The `opccallback` function is a user-defined AutoMod function that you can define to perform

processing when any subscribed item value changes (for more information, see “Responding to changes in subscribed item values” on [page 2.37](#)).

---

**Note** The value of an AutoMod variable is only updated if the value provided by the server has a quality that is “Good” (192 or higher); however, the `opccallback` function is called regardless of the value’s quality. For more information about OPC quality values, see Appendix B, “OPC Communication Functions.”.

---

Immediately after you call the `OPCSubscribe` function, the model attempts to initialize the subscribed item by reading the item’s current value on the server. The model then sets the value of the item’s associated AutoMod variable (if defined) and calls the `opccallback` function defined for the item (if defined).

**Important** In some cases, the model might not successfully initialize subscribed item values due to the latency of the OPC server. To ensure that all item values are successfully synchronized with the server, call the `OPCRefreshSubscription` function after subscribing to the appropriate items. For more information, see “Refreshing item subscriptions” on [page 2.35](#).

---

**Note** You can only subscribe to items that were added to a group on the server. See “Adding items to a group” on [page 2.17](#).

---

For more information about subscribing to items, see “OPCSubscribe function” on [page B.14](#).

## Example - Subscribing to items

In this example, an arriving procedure (`P_Subscribe`) subscribes to three items on a server and all items in group `GRMotor`. The procedure associates each individual item subscription with a corresponding variable in AutoMod.

---

**Note** When referring to items on the server, you might need to include a server name or alias in the item reference, depending on whether or not items on different OPC servers share the same name. If an item name is unique, you can improve model performance by referencing the item name alone. If you include a server name in an item reference, the server and item names must be separated by commas and cannot include any spaces, as shown. For more information, see “Defining a reference to an item” on [page B.26](#).

---



```

begin P_Subscribe arriving
  call F_isSubError(OPCSubscribe("TestSvr,Tempt",
    "V_tmprt"), "Tempt")
    /* subscribe using server alias and item name */
  call F_isSubError(OPCSubscribe("RStatus", "V_Rstat"),
    "RStatus") /* subscribe using item name alone */
  call F_isSubError(OPCSubscribe("LCount", "V_Lcount"),
    "LCount")
  call F_isSubError(OPCSubscribe("GRMotor", null), "GRMotor")
    /* Subscribe to all items in group "GRMotor"; the items
       are not associated with AutoMod variables. */
  call OPCRefreshSubscription(null)
    /* initialize linked variables */
  send to die
end

```

The function passes the return value of each `OPCSubscribe` function to the user-defined function `F_isSubError`. The function `F_isSubError` prints an error message and stops the model if any subscription fails:

```

begin F_isSubError function
  if (PrmRetValue < 0) then begin
    print "Error subscribing to item or group: " PrmSubName
      to message
    print "Request returned error: " PrmRetValue to message
    terminate /* stop the model */
  end
  return false /* return value ignored */
end

```

## Refreshing item subscriptions

OPC servers automatically update a model with a new item value whenever the value of a subscribed item changes by a percentage that is greater than the deadband defined for the item's group (for more information, see "Adding items to a group" on [page 2.17](#)). You can also automatically update the value of subscribed items at any point during a model run in two ways:

- You can call a function to refresh item subscriptions.
- You can dynamically refresh item subscriptions in the OPC Connections dialog box during a model run.

### Calling a function to refresh item subscriptions

To refresh item subscriptions at the same point during each model run, call the `OPCRefreshSubscription` function in the model. The function requires one string parameter that identifies the item you want to refresh. If you pass a `null` value in the parameter, the function automatically refreshes all subscribed item values. For more information, see "OPCRefreshSubscription function" on [page B.13](#).

To refresh an item value, the model reads the item's current value from its OPC server and updates the variable associated with the item in the model (for information about how to associate an AutoMod variable with an item, see "OPCSubscribe function" on [page B.14](#)). The model also automatically calls the `opccallback` function for each subscribed item that is refreshed.

**Example - Calling a function to refresh item subscriptions** In this example, the `P_RefreshMachs` arriving procedure calls the `OPCRefreshSubscription` function to refresh the value of three subscribed items.

```
begin P_RefreshMachs arriving procedure
  /* function identifies items using a server and item name */
  set V_index to 1 /* V_index is a variable of type Integer */
  while V_index <= 3 do begin
    print "TestSvr,Mach" V_index to V_ItemRef
    /* V_ItemRef is a variable of type String */
    call OPCRefreshSubscription(V_ItemRef)
    increment V_index by 1
  end
  print "Machines refreshed" to message
  send to die
end
```

The function increments an index to refresh the subscribed values `Mach1`, `Mach2`, and `Mach3`. When the `OPCRefreshSubscription` function is called, each variable that is associated with a subscribed item on the OPC server is automatically set to the value of the item on the server. The `opccallback` function (not shown) is automatically called for each subscribed item.

## Dynamically refreshing item subscriptions during a model run

You can update all variables associated with a subscribed item at any point during a model run in the OPC Connections dialog box. Refreshing item subscriptions in the OPC Connections dialog box also calls the `opccallback` function for each item that is refreshed.

### ► To dynamically refresh item subscriptions

1. In the Simulation window, select **OPC Connections** from the **Loads/Variables** menu.

The OPC Connections dialog box opens.

2. Click **Refresh Subscriptions**.

All subscribed item values are automatically read from the server, and the `opccallback` function is called for each refreshed item.

---

**Note** The OPC Connections dialog box also displays current item values (see "Displaying item values and update statistics" on [page 2.46](#)).

---

## Responding to changes in subscribed item values

Control systems often write item values that devices read to determine their behavior in the system. During emulation, an AutoMod model can respond to changes in item values in the same way as a device in the real system (for example, the control system might write item values to stop a conveyor section or stop a machine from processing). When a model is updated with a new value, the model can respond in the same way as the device (for example, by taking down a conveyor motor or by taking down a resource).

---

**Note** If you call functions to read item values at specific points during a simulation, you can write code to respond to the updated item values at the same time they are read. For more information, see “Example - Reading item values” on [page 2.31](#).

---

To respond to changes in subscribed item values in a model, define an `opccallback` function to indicate what actions the model should perform based on the new values. If defined, an `opccallback` function is automatically called whenever a subscribed item value changes by a percentage that is greater than the deadband defined for the items' group (see “Group properties” on [page 2.6](#)). The function is called regardless of whether or not an AutoMod variable is associated with the subscribed item (for more information, see “Subscribing to groups and items” on [page 2.33](#)).

You can define one `opccallback` function to respond to all changes in subscribed item values. You can also define separate functions for a specific machine, server, group, or subscribed item. If multiple functions apply to the same item (for example, a function is defined for both the item's server and for the item's group), only one function is called; the model determines which function to call according to function precedence. For information about function precedence and the syntax used to define an `opccallback` function, see “`opccallback` function” on [page B.23](#).

Each `opccallback` function has four system-defined parameters that you can use for determining how the model responds to a changed item value:

| Parameter                 | Description   |
|---------------------------|---|
| <code>theItem</code>      | A string that identifies the name of the item that has changed in value; the item's machine, server, and group names are <i>not</i> included in the string.   |
| <code>theData</code>      | <p>A string that identifies the current (updated) value of the item on the OPC server.</p> <p><b>Tip</b> You can cast an item's value from a string to an AutoMod data type (for example, a pointer or a numeric data type). To cast the value, use the <code>set</code> action in the model to set a variable of the appropriate type to the value of the parameter <code>theData</code>. The example on the following page shows how to cast an item's value to a <code>MotorPtr</code> in the model.</p> |
| <code>theTimestamp</code> | A string that defines the date and time when the item value changed on the server.  |
| <code>theQuality</code>   | An integer that indicates the quality of the value that was read from the server. For more information on quality codes and their meanings, see Appendix A, "OPC Quality Codes."  |

## Example - Responding to changes in subscribed item values

This example defines three `opccallback` functions:

- **An `opccallback` function for the group `GRMotor`.** Each Boolean item in this group represents the status of a conveyor section motor. Whenever the control system changes the value of an item to true or false, the function brings up or takes down a corresponding motor in the model.
- **An `opccallback` function for the item `Tempprt`.** This item indicates the temperature inside a tank. The `opccallback` function calls a different user-defined function, depending on whether the temperature inside the tank is increasing or decreasing.
- **A global `opccallback` function for all other subscribed items.** This function prints a message when a callback is received and checks the quality of the new item value, but does not perform any other processing based on the value.

Each callback function calls the user-defined functions `F_CountCBacks`, which counts the total number of callbacks, and `F_isQualityError`, which determines the quality of the value that is read from the server. If the quality of a subscribed item value is not "good," each callback function ends without processing the new value.

The software calls the following `opccallback` function for items in the group `GRMotor`:

```
begin opccallback "GRMotor" function
  /* defined for all items in group GRMotor */
  call F_CountCBacks(theItem)
  /* count total number of callbacks */
  if (F_isQualityError(theItem) is true) then
    return 0 /* return value ignored */

  print "conv:" theItem to V_MotorName
  /* Assuming each item name matches a motor name, create a
    string to reference the motor in AutoMod software. */
  set V_MotorPtr to V_MotorName
  /* cast the string to a MotorPtr */

  if (theData is "1") then
    bring up V_MotorPtr
  else
    take down V_MotorPtr
  return 0 /* return value ignored */
end
```

The function creates a pointer to a motor in the model by printing a string that consists of the conveyor system name, a colon, and the motor name (which is the same as the item name). The function then uses the `set` action to cast the string to a variable of type `MotorPtr`. After creating a pointer to the motor that corresponds to the subscribed item, the function brings up the motor if the item value is `true` (1) or takes down the motor if the item value is `false` (any other value).

The software calls the following `opccallback` function for the item `Temprt`:

```
begin opccallback "Temprt" function
  /* defined only for the item "Temprt" - highest precedence */
  call F_CountCBacks(theItem)
  /* count total number of callbacks */
  if (F_isQualityError(theItem) is true) then
    return 0 /* return value ignored */

  set V_Temprt to theData
  /* V_Temprt is a variable of type Real; the parameter's
    string type is automatically cast to real */
  if (V_OldTemp < V_Temprt) then call F_TempIncrease(V_Temprt)
  else call F_TempDecrease(V_Temprt)
  set V_OldTemp to V_Temprt
  /* reset the old temperature to the current value for
    future comparisons */
  return 0 /* return value ignored */
end
```

The function uses the `set` action to convert the item value from a string (stored in the parameter `theData`) to a real value (stored in the variable `V_Temprt`). After converting the value, the function compares

the new value to the most recent temperature (stored in the variable `V_OldTemp`) and calls a separate user-defined function depending on whether the temperature has increased or decreased.

The software calls the following global `opccallback` function for all other items:

```
begin opccallback function
  /* defined for all subscribed items - lowest precedence */
  call F_CountCBacks(theItem)
  /* count total number of callbacks */
  call F_isQualityError(theItem) /* check the item quality */
  return 0 /* return value ignored */
end
```

The `F_CountCBacks` user-defined function counts callbacks:

```
begin F_CountCBacks function
  increment C_callbacks current by 1
  /* C_callbacks is a counter */
  print "Received callback " C_callbacks total as .0 " for
    item " theItem to message
  return 0 /* return value ignored */
end
```

The `F_isQualityError` user-defined function checks each item's quality:

```
begin F_isQualityError function
  if (OPCQuality < 192) then begin
    print "Warning - Callback " C_callbacks total as .0 " for
      item " theItem "returned a quality value of "
        OPCQuality ". Ignoring callback." to message
    return true /* indicates an error with item quality */
  end
  return false /* indicates "good" quality */
end
```

---

## Running the model

There are important differences between running an emulation model and a stand-alone simulation model. Most notably, the model must be synchronized with the control system(s) you are testing. To synchronize a model with one or more control systems, AutoMod provides system-defined functions that you can call to:

- Synchronize the model's runtime rate with a control system.
- Signal the control system when the simulation is paused or continuing.
- Delay the simulation for a period of time measured in real time.
- Calculate the amount of real time that passes between two model events.

## Synchronizing a model's runtime rate with a control system

AutoMod models have the capability to run at rates much faster than real time. However, when you are emulating a material handling system, the speed at which a model can run is limited by the control system(s) you are testing. Some control systems cannot run faster than real time. Others, particularly software PLCs, can support faster runtime rates. To synchronize your model with one or more control systems, limit the model's runtime rate, and the rate of all control systems, to the rate of the slowest control system you are testing.

To define the model runtime rate, call the `SetSyncRate` function at the beginning of a simulation. For more information, see "SetSyncRate function" in the AutoMod online help.

### Example 1 - Limiting a model's runtime rate to real time

In this example, the `SetSyncRate` function limits the model's maximum runtime rate to real time. The model can run five seconds outside of the synchronization rate before calling the `model delayed` function.

```
begin model initialization function
    call SetSyncRate(1.0, 5.0)
    return true
end
```

The `model delayed` function is defined to print warnings to the Messages dialog box if the simulation runs outside the defined synchronization rate.

```
begin model delayed function
    if theDelay > 0 then begin /* simulation can't keep up */
        print "warning: The simulation has fallen " theDelay "
            seconds behind the defined synchronization rate of
            " syncrate " seconds. Increasing animation step."
            to message
    end
    else /* simulation could update graphics more frequently */
        print "warning: The simulation has delayed " theDelay "
            seconds to maintain synchronization. Decreasing
            animation step." to message
    end
    return true
end
```

### Example 2 - Limiting a model's runtime rate to twice the real time rate

In this example, the `SetSyncRate` function limits the model's maximum runtime rate to twice as fast as real time. The model can run slower than this rate for five seconds before stopping the simulation.

```
begin model initialization function
    call SetSyncRate(2.0, 5.0)
    return true
end
```

The `model delayed` function is defined to print warnings to the Messages dialog box if the simulation runs outside the defined synchronization rate.

```
begin model delayed function
  if theDelay > 0 then begin /* simulation can't keep up */
    print "Warning: The simulation has fallen " theDelay "
      seconds behind the defined synchronization rate of
      " syncrate " seconds. Increasing animation step."
    to message
  end
  else /* simulation could update graphics more frequently */
    print "Warning: The simulation has delayed " theDelay "
      seconds to maintain synchronization. Decreasing
      animation step." to message
  return true
end
```

### Example 3 - Limiting a model's runtime rate to half the real time rate

In this example, the `SetSyncRate` function limits the model's maximum runtime rate to half as fast as real time. The model can run slower than this rate for five seconds before stopping the simulation.

```
begin model initialization function
  call SetSyncRate(.5, 5.0)
  return true
end
```

The `model delayed` function is defined to print warnings to the Messages dialog box if the simulation runs outside the defined synchronization rate.

```
begin model delayed function
  if theDelay > 0 then begin /* simulation can't keep up */
    print "Warning: The simulation has fallen " theDelay "
      seconds behind the defined synchronization rate of
      " syncrate " seconds. Increasing animation step."
    to message
  end
  else /* simulation could update graphics more frequently */
    print "Warning: The simulation has delayed " theDelay "
      seconds to maintain synchronization. Decreasing
      animation step." to message
  return true
end
```



## Signaling a control system when paused or continuing

Pausing an emulation model is equivalent to temporarily stopping all emulated devices in the system. Because the control system you are testing continues to operate while a simulation is paused, the OPC server continues to update the simulation with changes in subscribed item values. The simulation's delay in responding to these values might cause errors in the control system.

---

**Note** While the simulation is paused, AutoMod stores updated values in a buffer; if the simulation is paused for a long period of time, the item updates can result in an out-of-memory error.

---

To prevent these errors, AutoMod provides two functions that you can use to signal the control system(s) you are testing when you pause or continue a simulation:

| Function                    | Description   |
|-----------------------------|---|
| <code>model pause</code>    | This function is automatically called whenever you pause the simulation; either by pressing <b>P</b> or by selecting <b>Pause</b> from the <b>Control</b> menu.       |
| <code>model continue</code> | This function is automatically called whenever you continue the simulation; either by pressing <b>P</b> or by selecting <b>Continue</b> from the <b>Control</b> menu. |

You can define these functions to perform any processing necessary to notify a control system when the model pauses or continues. For example, you can change the value of a Boolean item on the OPC server to true or false; the control system can use this item value to respond to the simulation delay.

**Important** Interacting with simulation windows (for example, checking statistics or displaying business graphs) temporarily delays the simulation. For this reason, you should always pause the model by pressing **P** before interacting with the simulation.

### Example - Writing an item value when a server pauses or continues

In this example, the `model pause` function sets the value of the Boolean item `Pause` to 1 (`true`) whenever the simulation pauses. If the write fails, the function calls the user-defined function `F_WriteFailed` (not shown).

```
begin model pause function
  set V_WriteResult to OPCWriteIntegerWait("Pause", 1, 1.0)
  /* toggle the Pause item value to true */
  if V_WriteResult < 0 then call F_WriteFailed("Pause")
end
```

Similarly, the `model continue` function sets the value of the Boolean item `Pause` to 0 (false) whenever the simulation continues:

```
begin model continue function
  set V_WriteResult to OPCWriteIntegerWait("Pause", 0, 1.0)
  /* toggle the Pause item value to false */
  if V_WriteResult < 0 then call F_WriteFailed("Pause")
end
```

Using this approach, the control system can subscribe to the `Pause` item value to receive automatic updates whenever the simulation pauses or continues.

## Creating delays measured in real time

To communicate with a control system, you might need to create time delays that are measured in real time. For example, some control systems or OPC servers require a small delay between received messages to prevent data loss. You can create such a delay by calling the `MilliSleep` function after sending a message. You can use the `MilliSleep` function to create any real time delay that is required during a model run.

---

**Note** You cannot create time delays in a function; you can only call the `MilliSleep` function within a procedure or subroutine.

---

The `MilliSleep` function requires one parameter, which defines the length of the time delay in milliseconds. The actual time paused is accurate to within 10 milliseconds.

### Example - Creating a delay between messages

In this example, the arriving procedure `P_WriteStats` writes three values to the OPC server. The `MilliSleep` function is used to add a half-second delay (500 milliseconds) between each message.

```
begin P_WriteStats arriving procedure
  set V_IVInt to tp:tank1 capacity
  call OPCWriteReal("TCap",V_IVInt)
  call MilliSleep(500)

  set V_IVReal to tp:tank1 level
  call OPCWriteReal("TLev",V_IVReal)
  call MilliSleep(500)

  set V_IVReal to tp:tank1 total rate in
  call OPCWriteReal("TRatIn",V_IVReal)
  call MilliSleep(500)

  send to P_next
end
```

## Calculating the real time that passes between model events

When validating your model, it might be useful to measure the amount of real time that passes between two model events (for example, you can calculate the average message response time or the length of the model run). To measure the time between events in real time you can set variables to the real time clock using the system keyword `realclock`, and then compare the variable values to determine how much time has passed.

The keyword `realclock` returns the number of seconds since midnight (00:00:00), January 1, 1970, coordinated universal time (UTC). The time value is accurate to within .01 seconds.

### Example - Calculating message response time

In this example, the `F_TestMsgTime` function writes a value to a subscribed item, `ITest`, on an OPC server. The procedure sets the real variable `V_MsgStart` to the time of the real clock when the message is sent. The callback function defined for the item `ITest`, sets the value of another real variable, `V_MsgFinish`, to the real clock when a callback is received from the server indicating that the item's value has changed. By subtracting the first time from the second, the function determines the message's response time, which is printed to the Messages dialog box.

```
begin F_TestMsgTime function
  call OPCWriteInteger("ITest", 1)
  set V_MsgStart to realclock
  return true
end

begin opccallback "ITest" function
  set V_MsgFinish to realclock
  print "The message response time is:
    " V_MsgFinish - V_MsgStart to message
  return true
end
```

---

## Viewing statistics and communication information after a run

After a model run is complete, you can review statistics for the run by opening the AutoMod report (the AutoMod report is named `<modelname>.report` and is saved in the model directory). For information about how to interpret the AutoMod report, see "Viewing the AutoMod report" in the "Running a Model" chapter in the *AutoMod User's Guide*.

AutoMod provides OPC specific statistics in a display dialog box in the software. By opening this dialog box, you can view item values and update statistics gathered during the model run. You can also create a log of all the values passed between the model and the OPC server during the model run by creating a trace file. Trace files can also be

used to duplicate the results of a model run without connecting to a live OPC server. These options are discussed in more detail in the following sections.

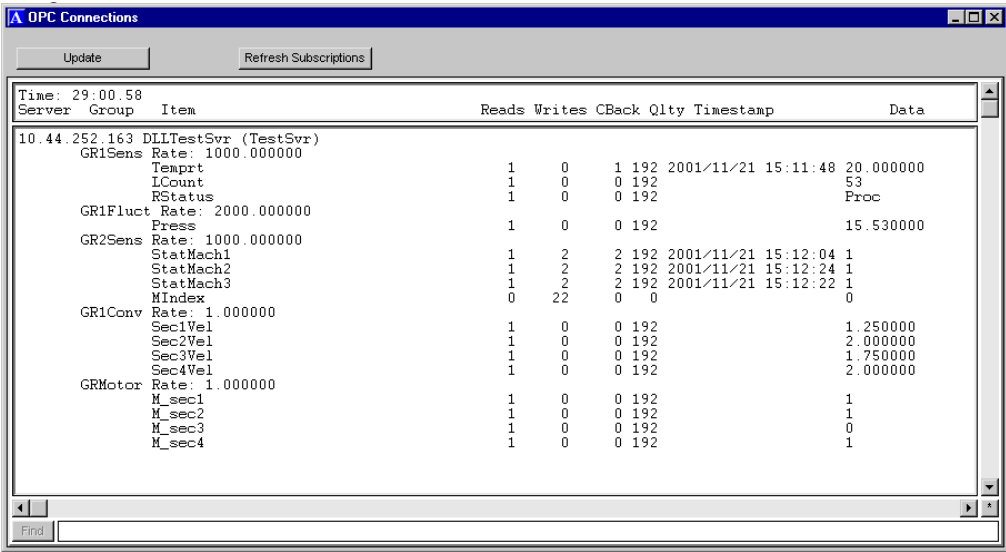
## Displaying item values and update statistics

**Important** You can display item values and update statistics at any point during a simulation; however, interacting with AutoMod during a run temporarily delays the model and could cause errors in the control system you are testing. If you have defined the model pause and model continue functions to notify the control system when the simulation is paused, then press **P** to pause the model before viewing statistics. Otherwise, only view statistics after the run is complete.

► **To display item values and statistics**

In the Simulation window, select **OPC Connections** from the **Loads/Variables** menu.

The OPC Connections dialog box opens as shown:



**Note** Information in the OPC Connections dialog box is not dynamically updated if the dialog box is open while the model is running. You can click Update to update the information in the dialog box to the current clock time.

Information provided for each item is defined as follows:

| Value  | Description   |
|--------|---|
| Server | The name of the OPC server to which the model is connected. |

| Value     | Description   |
|-----------|---|
| Group     | The name of a group that is defined on the server.  |
| Item      | The name of an item on the server.  |
| Reads     | <p>The number of times the model has read this item value from the server.</p> <p><b>Note</b> When you subscribe to an item on an OPC server, the model initializes the subscribed value in the model by reading the item value, which increments the Reads statistic by 1.</p>   |
| Writes    | The number of times the model has written a value for this item to the server.  |
| CBack     | <p>The number of times the <code>opccallback</code> function has been called as a result of updating this item.</p> <p><b>Note</b> The CBack statistic does not include the first callback that the software executes to initialize a subscription to an item. To initialize the value of subscribed items, the software reads the item value, which increments the read statistic rather than the CBack statistic.</p> |
| Qlty      | The integer quality code returned by the server for the last value of this item read by the model. For more information about OPC quality codes and their meanings, Appendix A, “OPC Quality Codes.”  |
| Timestamp | <p>The date and time that the displayed item value was read from the server.</p> <p><b>Note</b> The Timestamp is not displayed for values that the model reads to initialize subscribed items.</p>  |
| Data      | The value of this item that was last read from the server.  |

## Using trace files

A *trace file* is a tab-delimited text file that contains a list of the communication events that occur during a model run. A communication event is an action or occurrence that the AutoMod model or the OPC server can detect. For example, communication events include all values that were read from and written to the server. Communication events also include function calls that create groups on the server, add items to groups, subscribe to items, and so on.

Trace files are useful for debugging. A trace file allows you to see the time at which events occurred during the run. Trace files are also useful for repeating a model run, without connecting to an OPC server. When you run a model using a trace file, the run reproduces the traced run by reading changes in item values from the trace file, rather than an OPC server.

### Creating a trace file during a model run

#### ► To create a trace file during a model run

1. In AutoMod, build the model.
2. Open a command prompt and navigate to the directory of the model executable.
3. At the command prompt, type the following:  
`<model_executable_name> -OPC  
tracefile=<trace_file_name>,trace=<trace_level>`

---

**Note** The trace file name must be separated from the trace command with a comma and must not contain any spaces.

---

If you omit a command when starting the model, the software uses the default value for the command. The commands that create a trace file are defined as follows:

**trace\_file\_name** The name of a text file to which messages are written during the model run. The trace file is created in the model directory. If a file with the same name already exists, it is overwritten. If the file doesn't already exist, it is created automatically.

Default value: opc.trace

**trace\_level**An integer that determines which messages are written to the trace file. The following table lists the valid integer values and provides a description of each:

| Trace level | Description  |
|-------------|--|
| 0 (default) | A trace file is not created.   |
| 1           | The trace file includes all communication events except for those that write values to the OPC server. |
| 2           | The trace file includes all communication events, including those that write values to the OPC server. |

Default value: 0 (messages are not traced)

For example, to create a level-two trace file for the model `demo1`, type:

```
demo1 -OPC tracefile=trace.txt,trace=2
```

During the model run, communication events are recorded in the trace file `trace.txt`.

### Trace file format

Trace files are ASCII text files that you can open in any text editor. Reviewing the information in a trace file is often useful for debugging.

The first line of a trace file displays the trace level. The second line of the trace file displays the headings for each column of data in the file. The file contains the following headings:

| Heading | Description  |
|---------|--|
| OPCType | <p>The type of communication event that occurred. The possible types are:</p> <ul style="list-style-type: none"> <li>• <code>OPC_ADDGROUP</code> - Creates a group on the OPC server; this event occurs when the model calls the <code>OPCAddGroup</code> function.</li> <li>• <code>OPC_ADDITEM</code> - Adds an existing item on the server to a group; this event occurs when the model calls the <code>OPCAddItem</code> function.</li> <li>• <code>OPC_CALLBACK</code> - Updates the value of a subscribed item in the model; this event occurs when a subscribed item's value changes by a percentage that is greater than the deadband defined for the item's group (see "Adding items to a group" on <a href="#">page 2.17</a>). If there is an AutoMod variable associated with the subscribed item, the variable's value is automatically updated to the subscribed item's value. When this event occurs, the model automatically calls the <code>opccallback</code> function for the item.</li> <li>• <code>OPC_CONNECT</code> - Opens a connection between the model and the OPC server; this event occurs when the model calls the <code>OPCConnectServer</code> function.</li> <li>• <code>OPC_DISCONNECT</code> - Closes a connection to the OPC server; this event occurs when the model calls the <code>OPCDisconnect</code> function or when you close the AutoMod runtime environment.</li> <li>• <code>OPC_READ</code> - Reads the current value of an item on the OPC server; this event occurs when the model calls the <code>OPCReadString</code>, <code>OPCReadInteger</code>, or <code>OPCReadReal</code> function.</li> <li>• <code>OPC_WRITE</code> - Sets the value of an item on an OPC server; this event occurs when the model calls the <code>OPCWriteString</code>, <code>OPCWriteInteger</code>, or <code>OPCWriteReal</code> function.</li> <li>• <code>OPC_SERVERDEAD</code> - Terminates the model; this event occurs when the server has gone down and the connection is no longer valid.</li> </ul> |



| Heading   | Description   |
|-----------|---|
|           | <ul style="list-style-type: none"> <li>• <code>OPC_SUBSCRIBE</code> - Subscribes to an item or group on the OPC server; this event occurs when the model calls the <code>OPCSubscribe</code> function.</li> <li>• <code>OPC_WRITE</code> - Sets the value of an item on an OPC server; this event occurs when the model calls the <code>OPCWriteString</code>, <code>OPCWriteInteger</code>, or <code>OPCWriteReal</code> function.</li> <li>• <code>OPC_SERVERDEAD</code> - Terminates the model; this event occurs when the server has gone down and the connection is no longer valid.</li> <li>• <code>OPC_SUBSCRIBE</code> - Subscribes to an item or group on the OPC server; this event occurs when the model calls the <code>OPCSubscribe</code> function.</li> </ul> |
| Time      | The simulation time when the event occurred. The time is expressed in model default units.  |
| Timestamp | The date and time when the event occurred; the timestamp is provided by the OPC server for <code>OPC_READ</code> and <code>OPC_CALLBACK</code> events.  |
| Server    | The name of the OPC server.   |
| Group     | The name of the group on the server.  |
| Item      | The name of the item on the server.   |
| Data      | The value of the item (either written to or read from the server).  |
| Status    | The integer quality code provided by the server for the item's value. For more information about OPC quality codes and their meanings, see Appendix A, "OPC Quality Codes." If the communication event is not associated with an item value (for example <code>OPC_CONNECT</code> , <code>OPC_ADDGROUP</code> , and so on), the Status field provides the return value of the internal function call that created the event.  |

## Example - Trace file format

The following example shows a level-two trace file from a model:

| OPC TraceFile |          | TraceLevel 2           |            |          |           |           |          |
|---------------|----------|------------------------|------------|----------|-----------|-----------|----------|
| OPCType       | Time     | Timestamp              | Server     | Group    | Item      | Data      | Status   |
| OPC_CONNECT   | 0.000000 |                        | DLLTestSvr |          |           |           | 60688096 |
| OPC_ADDGROUP  | 0.000000 |                        | TestSvr    | GR1Sens  |           |           | 12391312 |
| OPC_ADDGROUP  | 0.000000 |                        | TestSvr    | GR1Fluct |           |           | 12391024 |
| OPC_ADDGROUP  | 0.000000 |                        | TestSvr    | GR2Sens  |           |           | 12390736 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR1Sens  | Temprt    |           | 12390512 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR1Sens  | LCount    |           | 12398496 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR1Sens  | RStatus   |           | 12398400 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR1Fluct | Press     |           | 12398304 |
| OPC_ADDGROUP  | 0.000000 |                        | TestSvr    | GR1Conv  |           |           | 12398160 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR1Conv  | Sec1Vel   |           | 12397984 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR1Conv  | Sec2Vel   |           | 12397872 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR1Conv  | Sec3Vel   |           | 12397744 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR1Conv  | Sec4Vel   |           | 12397648 |
| OPC_ADDGROUP  | 0.000000 |                        | TestSvr    | GRMotor  |           |           | 12397504 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GRMotor  | M_sec1    |           | 12397296 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GRMotor  | M_sec2    |           | 12397184 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GRMotor  | M_sec3    |           | 12397056 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GRMotor  | M_sec4    |           | 12396960 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR2Sens  | StatMach1 |           | 12396864 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR2Sens  | StatMach2 |           | 12396768 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR2Sens  | StatMach3 |           | 12396640 |
| OPC_ADDITEM   | 0.000000 |                        | TestSvr    | GR2Sens  | MIndex    |           | 12396544 |
| OPC_SUBSCRIBE | 0.000000 |                        | TestSvr    | GR1Sens  | Temprt    |           | 1        |
| OPC_READ      | 0.000000 | 2001/12/03<br>15:41:53 | TestSvr    | GR1Sens  | Temprt    | 19.200001 | 192      |
| OPC_SUBSCRIBE | 0.000000 |                        | TestSvr    | GR1Sens  | RStatus   |           | 1        |
| OPC_READ      | 0.000000 | 2001/12/03<br>15:41:53 | TestSvr    | GR1Sens  | RStatus   | Proc      | 192      |
| OPC_SUBSCRIBE | 0.000000 |                        | TestSvr    | GR1Sens  | LCount    |           | 1        |
| OPC_READ      | 0.000000 | 2001/12/03<br>15:41:53 | TestSvr    | GR1Sens  | LCount    | 53        | 192      |
| OPC_SUBSCRIBE | 0.000000 |                        | TestSvr    | GR1Fluct |           |           | 1        |
| OPC_READ      | 0.000000 | 2001/12/03<br>15:41:53 | TestSvr    | GR1Fluct | Press     | 15.530000 | 192      |

| OPC TraceFile |             | TraceLevel 2           |            |         |           |          |        |
|---------------|-------------|------------------------|------------|---------|-----------|----------|--------|
| OPCType       | Time        | Timestamp              | Server     | Group   | Item      | Data     | Status |
| OPC_SUBSCRIBE | 0.000000    |                        | TestSvr    | GR2Sens | MIndex    |          | 1      |
| OPC_READ      | 0.000000    | 2001/12/03<br>15:41:53 | TestSvr    | GR2Sens | MIndex    | 1        | 192    |
| OPC_SUBSCRIBE | 0.000000    |                        | TestSvr    | GRMotor |           |          | 1      |
| OPC_READ      | 0.000000    | 2001/12/03<br>15:41:53 | TestSvr    | GRMotor | M_sec1    | 1        | 192    |
| OPC_READ      | 0.000000    | 2001/12/03<br>15:41:53 | TestSvr    | GRMotor | M_sec2    | 1        | 192    |
| OPC_READ      | 0.000000    | 2001/12/03<br>15:41:53 | TestSvr    | GRMotor | M_sec3    | 0        | 192    |
| OPC_READ      | 0.000000    | 2001/12/03<br>15:41:53 | TestSvr    | GRMotor | M_sec4    | 1        | 192    |
| OPC_READ      | 0.000000    | 2001/12/03<br>15:41:53 | TestSvr    | GR1Conv | Sec1Vel   | 1.250000 | 192    |
| OPC_READ      | 0.000000    | 2001/12/03<br>15:41:53 | TestSvr    | GR1Conv | Sec2Vel   | 1.500000 | 192    |
| OPC_READ      | 0.000000    | 2001/12/03<br>15:41:53 | TestSvr    | GR1Conv | Sec3Vel   | 1.750000 | 192    |
| OPC_READ      | 0.000000    | 2001/12/03<br>15:41:53 | TestSvr    | GR1Conv | Sec4Vel   | 2.000000 | 192    |
| OPC_CALLBACK  | 762.001000  | 2001/12/03<br>15:42:18 | DLLTestSvr | GR2Sens | MIndex    | 0        | 192    |
| OPC_WRITE     | 839.487335  |                        | TestSvr    | GR2Sens | StatMach1 | 0        | 1      |
| OPC_WRITE     | 914.104726  |                        | TestSvr    | GR2Sens | StatMach1 | 1        | 1      |
| OPC_CALLBACK  | 914.105000  | 2001/12/03<br>15:42:22 | DLLTestSvr | GR2Sens | MIndex    | 1        | 192    |
| OPC_WRITE     | 1396.213711 |                        | TestSvr    | GR2Sens | StatMach3 | 0        | 1      |
| OPC_WRITE     | 1428.924032 |                        | TestSvr    | GR2Sens | StatMach3 | 1        | 1      |
| OPC_CALLBACK  | 1462.001000 | 2001/12/03<br>15:42:34 | DLLTestSvr | GR2Sens | MIndex    | 0        | 192    |
| OPC_WRITE     | 1475.063157 |                        | TestSvr    | GR2Sens | StatMach2 | 0        | 1      |
| OPC_WRITE     | 1512.105608 |                        | TestSvr    | GR2Sens | StatMach2 | 1        | 1      |
| OPC_CALLBACK  | 1618.001000 | 2001/12/03<br>15:42:38 | DLLTestSvr | GR2Sens | MIndex    | 1        | 192    |
| OPC_WRITE     | 1885.245307 |                        | TestSvr    | GR2Sens | StatMach3 | 0        | 1      |
| OPC_WRITE     | 1925.319694 |                        | TestSvr    | GR2Sens | StatMach3 | 1        | 1      |
| OPC_CALLBACK  | 2064.001000 | 2001/12/03<br>15:42:46 | DLLTestSvr | GR2Sens | MIndex    | 0        | 192    |
| OPC_CALLBACK  | 2362.873000 | 2001/12/03<br>15:42:50 | DLLTestSvr | GR2Sens | MIndex    | 1        | 192    |
| OPC_WRITE     | 2585.511260 |                        | TestSvr    | GR2Sens | StatMach3 | 0        | 1      |

| OPC TraceFile  |             | TraceLevel 2           |            |         |           |      |        |
|----------------|-------------|------------------------|------------|---------|-----------|------|--------|
| OPCType        | Time        | Timestamp              | Server     | Group   | Item      | Data | Status |
| OPC_WRITE      | 2677.545484 |                        | TestSvr    | GR2Sens | StatMach3 | 1    | 1      |
| OPC_CALLBACK   | 2767.000000 | 2001/12/03<br>15:42:58 | DLLTestSvr | GR2Sens | MIndex    | 0    | 192    |
| OPC_WRITE      | 2862.410199 |                        | TestSvr    | GR2Sens | StatMach2 | 0    | 1      |
| OPC_WRITE      | 2904.872914 |                        | TestSvr    | GR2Sens | StatMach2 | 1    | 1      |
| OPC_CALLBACK   | 2940.000000 | 2001/12/03<br>15:43:02 | DLLTestSvr | GR2Sens | MIndex    | 1    | 192    |
| OPC_CALLBACK   | 3465.000000 | 2001/12/03<br>15:43:08 | DLLTestSvr | GR2Sens | MIndex    | 0    | 192    |
| OPC_DISCONNECT | 3600.000000 |                        | TestSvr    |         |           |      | 0      |

The communication events at time 0 open the connection to the OPC server and set up the groups and items that the model can access throughout the run. After subscribing to an item or group, the model automatically reads the initial value of each subscribed item. The model also reads conveyor velocity values from the OPC server to initialize the conveyor sections in the model.

Throughout the model run, the model receives updates from the OPC server to take down or bring up the indexing conveyor motor, `MIndex`. The model writes values to the OPC server to indicate the failure and repair of the three `StatMach` machines.

At the end of the simulation (3600 seconds), the model disconnects from the OPC server.

## Running a model with a trace file

Running a model with a trace file allows you to reproduce a traced run without connecting to any OPC servers. Running a model with a trace file is useful when you want to duplicate a model run, but a connection to one or more OPC servers is not available. To run a model with a trace file, the level of the trace file must be greater than zero (for more information, see “Creating a trace file during a model run” on [page 2.48](#)).

**Important** If you make changes to a control system or to the model after tracing a run, you must create a new trace file for accurate results.

### ► To run a model with a trace file

1. Create the trace file (for more information, see “Creating a trace file during a model run” on [page 2.48](#)).
2. Open a command prompt and navigate to the directory of the model executable.

3. At the command prompt, type the following:

```
<model_executable_name> -OPC file=<trace_file_name>
```

For example, to start the model `demo1` using the trace file `trace.txt`, type:

```
demo1 -OPC file=trace.txt
```

The model runs exactly as it did during the traced run.

---

## Testing scenarios

After completing the emulation model, you are ready to begin testing the control system. One of the primary benefits of emulation is that you can test the control system using a wide range of scenarios. By altering load arrival rates, machine processing times, downtimes, and so on, you can test the control system's response to both anticipated and extreme conditions.

The easiest way to test various system scenarios is to run the emulation model in AutoStat™. AutoStat automates the process of testing multiple scenarios and provides advanced graphing and reporting features for analyzing model results. For information on how to define analyses and run a model in the AutoStat software, see the "AutoStat" chapter in the *AutoMod User's Guide*.



# Socket Communications

In this chapter, you will learn:

- How stream sockets communicate.
- How to open, verify, and close a socket connection.
- How to send messages.
- How to read and interpret received messages.
- How to synchronize two models without using the MMS Server.

---

## Introduction

This chapter describes how to open socket connections and send and read messages. A *message* is any data sent through a socket connection. Depending on its format, a message can consist of the same or different types of data (for example, *String*, *Real*, and *Integer*). Models can communicate through sockets on the same processor, on multiple processors running on the same machine, and/or on multiple machines connected by a network. This chapter also explains how to synchronize two models without using the MMS server.

**Important** If you have purchased and authorized the Model Communications Plus module, you do not need to open socket connections or send and read messages as described in this chapter. The MMS server automatically opens connections between synchronized models, and sends and reads messages of a different format than those described in this chapter. For information about communicating between multiple models that are synchronized by the MMS server, see Chapter 4, “Multi-Model Synchronization (Model Communications Plus module).”

---

## Managing socket connections

Models transfer data through stream sockets using the Transfer Control Protocol (TCP).

*Stream sockets* are a type of socket that can both send and receive data. Stream sockets send data in sequenced, unduplicated packets (the data is sent in the order it is provided, and is sent only once). A *packet* is a bundle of data that varies in size depending on your network hardware, but is usually less than 1500 bytes.

The *Transmission Control Protocol (TCP)* is a reliable byte-stream transfer service between sockets. TCP protects against data loss and data corruption. TCP also provides flow-control that continually adjusts the rate at which data is sent in an attempt to achieve maximum data throughput while avoiding network congestion and packet loss.

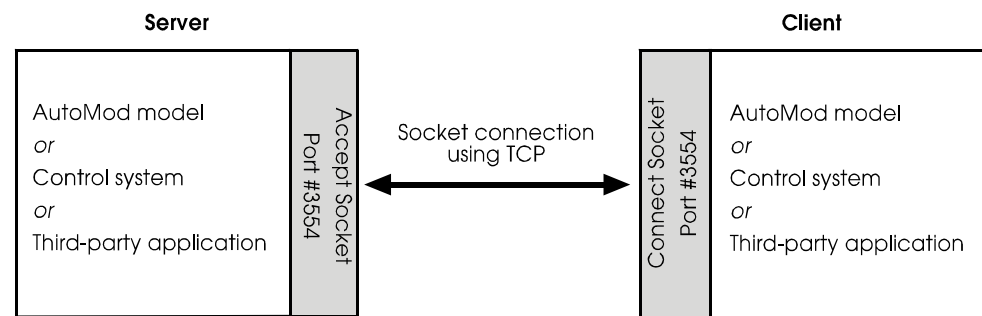
Managing socket connections consists of:

- Opening socket connections
- Verifying that connections are valid
- Closing connections during a simulation (in many cases, connections close automatically)

## Opening a connection

Sockets can transfer data on the same processor, between multiple processors running on the same machine, and/or between multiple machines connected by a network. Before models can send or read messages, you must open a socket connection.

Socket connections have a client/server relationship. The client sends a connection attempt and the server sends a connection acceptance. The server must receive the connection attempt and the client must receive the acceptance before a connection is established. The following illustration shows a socket connection between a server and client.



Opening a socket connection

Before attempting or accepting a connection, you must complete the following:

- Edit the hosts file (if not connected to a Domain Name System server).
- Define the length of connection attempts.
- Define a port for the socket connection.
- Determine the number of connections you want to open.

These steps are discussed in detail on the following pages.



## Editing the hosts file

Several of the functions that are used for model communications use machine names to locate the IP addresses of remote machines on a network. In most cases, machine names are mapped to their corresponding IP numbers by a Domain Name System server (DNS server) on your local area network.

---

**Note** If your machine is connected to a DNS server, you do not need to alter the hosts file as described in this section.

---

If you are not connected to a DNS server, you must edit the hosts file on your machine to manually map remote machine names to their IP addresses.

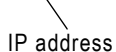
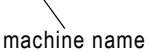

To find out if a DNS server is available on your network, or for information about obtaining the IP addresses of machines on your network, contact your network administrator.

### ► To edit the hosts file

1. Log in as administrator.
2. Using a text editor, edit the file  
%SystemRoot%\System32\drivers\etc\hosts.
3. The hosts file contains a list of remote machine names and their corresponding IP addresses. Add a line to the bottom of the file that matches the following format:

<IP\_address> <machine\_name> # <comment>

Here is an example:

|   |  |   |
|---|--|---|
| 102.54.94.97  | mymachine.mycompany.com  | # source server   |
|  |  |  |
| IP address  | machine name   | comment   |

The machine name `mymachine.mycompany.com` is now mapped to the IP address 102.54.94.97.

## Defining the length of connection attempts

The length of time a server or client attempts to open a socket connection depends on whether the server or client is set to blocking or timeout.

When a server or client is set to *blocking*, the simulation delays until a connection is established; the server's acceptances or the client's connection attempts continuously repeat until a connection is opened.

When a server or client is set to *timeout*, the server's acceptances or the client's connection attempts repeat for a specified length of time; if a connection is not opened within the defined time period, the server or client stops trying to open a connection.

---

**Tip** Timeout connections are useful when creating models that may be run independently, or that periodically check for connections during a simulation. Use blocking connections when a model cannot be run independently and must send and read messages for accurate simulation.

---

**Server** By default, a server's connection acceptances are set to blocking.

| To do this:                                       | Use this function:                             |
|---|--|
| Set a server's acceptances to timeout or blocking | <code>SetAcceptBlocking(&lt;Toggle&gt;)</code> |
| Define the length of timeout acceptances          | <code>SetAcceptTimeOut(&lt;Time&gt;)</code>    |

The parameter you pass to the `SetAcceptBlocking` function must be a 1 or 0 to indicate the server is set to blocking or timeout, respectively. If the server is set to timeout, the parameter you pass to the `SetAcceptTimeOut` function indicates the amount of time in seconds that the server attempts to accept a connection before continuing. Call these functions before the server accepts a connection. For more information, see appendix C, "Socket Communication Functions."

**Client** By default, a client's connection attempts are set to time out in zero seconds (if the server is not accepting at the exact time the client attempts to connect, the connection fails).

| To do this:   | Use this function:                              |
|---|---|
| Set a client's connection attempts to timeout or blocking | <code>SetConnectBlocking(&lt;Toggle&gt;)</code> |
| Define the length of timeout attempts                     | <code>SetConnectTimeOut(&lt;Time&gt;)</code>    |

The parameter you pass to the `SetConnectBlocking` function must be a 1 or 0 to indicate the client is set to blocking or timeout, respectively. If the client is set to timeout, the parameter you pass to the `SetConnectTimeOut` function indicates the amount of time in seconds that the client attempts to open a connection to the server, before

continuing. Call these functions before the client attempts to connect to a server. For more information, see appendix C, “Socket Communication Functions.”

**Important** Use caution when changing the server’s acceptances to timeout. If a client attempts to connect to a server after the server acceptance has timed out, the client does not recognize that the connection has failed until it tries to read or send a message using that socket connection (at which time the connection closes automatically).

### Example 1 - Defining timeout connections

This example shows how to define the length of connection attempts. The `model initialization` function is used in two models to set the client’s connection attempts and the server’s acceptances to a timeout of 5 minutes (300 seconds).

These functions are called before the client attempts a connection and before the server accepts a connection.

#### Server model

```
begin model initialization function
  call SetAcceptBlocking(0) /* server is set to timeout */
  call SetAcceptTimeout(300)
    /* server will accept connections for 300 seconds */
  return true /* return value not used */
end
```

#### Client model

```
begin model initialization function
  /* client is set to timeout by default */
  call SetConnectTimeout(300)
    /* client will attempt connections for 300 seconds */
  return true /* return value not used */
end
```

### Example 2 - Defining blocking connections

In this example, both the client and server models are set to blocking. When the client sends a connection attempt, the client’s simulation delays and the attempt repeats until a connection is opened. Likewise, when the server sends an acceptance, the server’s simulation delays and the acceptance repeats until a connection is opened.

#### Server model

```
begin model initialization function
  call SetAcceptBlocking(1) /* server is set to blocking */
  return true /* return value not used */
end
```

### Client model

```
begin model initialization function
    call SetConnectBlocking(1) /* client is set to blocking */
    return true /* return value not used */
end
```

### Defining a port for a socket connection

When opening a socket connection, you must define the port that you are going to use for that connection. *Ports* are defined by passing either a port number or a service name to the function that is used to attempt or accept a connection (the functions used for establishing connections are discussed later in this chapter).

A *port number* is a number greater than 1000 that is used by TCP to match the client's connection attempt with the server's acceptance.

#### Important

- The port numbers used for the client connection attempt and server acceptance must be the same.
- ! • Do not select a port number that has been reserved for another service on your machine; numbers less than 1000 are reserved for standard services, such as file transfer protocol (FTP). You can display a list of the reserved port numbers on your machine by opening the services file as described in the following section.

A *service name* is an alphabetic reference to a port number. For example, the service name `model1` can be used to represent port # 3554. If you want to use a service name, you must first define it in the services file. The steps for defining a service name are provided in the following section.

**Defining a service name** If you want to use a service name to open a connection, you must first edit the services file and define a service name and the port number it represents.

---

**Note** There is no difference between opening a connection using a port number or service name, but the service name may be easier to remember.

---

- ! • **Important** The port number must be the same on both the client and server, but the service name that refers to the port number does not need to match.

#### ► To define a service name

1. Log in as `administrator`.
2. Using a text editor, edit the file  
    `%SystemRoot%\System32\drivers\etc\services`.

3. The services file contains a list of services and reserved port numbers on your machine. Add a line to the bottom of the file that matches the following format:

`<service_name> <port_number>/tcp # <comment>`

Here is an example:

```
modell 3554/tcp # First model's socket connection
```

service name      port number      comment

The service name `modell` can now be used to refer to port number 3554.

## Opening multiple connections

You can open an unlimited number of socket connections between a client and server. However, the more open socket connections to your model, the more blocking and timeout delays are necessary to open connections and read messages during simulation.

Use the following guidelines to determine how many socket connections to open:

- The model and client must have at least one open socket connection through which to send and receive messages.
- If you are planning on sending and receiving both string and C structure messages, you should have a separate socket connection for each message format (see “Message formats” on [page 3.13](#)).

---

**Tip** You can open more than one socket connection using the same port. There is no difference between opening multiple socket connections using the same or different ports, but grouping connections by a shared port may make keeping track of connections easier. For example, you can open two socket connections (one for string messages and the other for C structure messages) using the same port, but use a different port for each model to which you are opening a connection. For more information, see “Defining a port’s number of accepted connections” below.

---

**Defining a port's number of accepted connections** By default, a server accepts only one connection per port. There are two AutoMod functions that allow you to increase the number of accepted socket connections for a particular port.

**To define a port's number of accepted connections using a:**      **Use this function:**

|              |  |
|--------------|--|
| Port number  | SetNumAcceptsPort(<Port_number>,<Connections>) |
| Service name | SetNumAcceptsService(<Service>,<Connections>)  |

These functions accept two parameters: the first parameter identifies either the port number (an integer) or the service name (a string), and the second parameter defines the number of connections the port can accept (an integer). These functions must be called before the server accepts a connection. For more information, see appendix C, "Socket Communication Functions."

**Tip** You can define an infinite number of allowed connections for a port by passing zero to either of the functions listed above.

**Note** It is only necessary to call these functions in the server model; you do not need to call them in the client model.

**Example - Defining multiple connection acceptances for a port** In this example, the `SetNumAcceptsPort` function sets the number of connections that can be accepted by the server on port number 5555 to 2.

```
begin model initialization function
  call SetNumAcceptsPort(5555, 2)
  return true /* return value not used */
end
```

## Attempting a connection (client)

There are two AutoMod functions you can use to attempt a connection to a server. Choose the function based on whether you want to open the connection using a service name or port number (see "Defining a port for a socket connection" on [page 3.6](#)).

**To attempt a connection**      **Set a variable of type SocketPtr using this function:**  
**using a:**

|              |   |
|--------------|---|
| Port number  | ConnectSocketPort(<Server>,<Port_number>) |
| Service name | ConnectSocketService(<Server>,<Service>)  |

The first parameter of both functions defines the server's machine name (a string). The second parameter identifies the port number (an integer) or service name (a string) to use when opening the connection. For more information, see appendix C, "Socket Communication Functions."

---

**Tip** If the server and client models are both running on the same machine, you can use the `MachineName` function to obtain the name of the server machine. The name of the server machine is passed as a parameter to both the `ConnectSocketPort` and `ConnectSocketService` functions. For more information, see "MachineName function" on [page C.6](#).

---

The return value of both functions is a socket pointer that points to the open connection; this pointer is used later to send and read messages using this socket connection.

---

**Tip** Attempt a connection in the `model_ready` function, because it is called at the beginning of a simulation. For more information, see "Time-specific functions" in the "Process system" chapter of the *AutoMod User's Guide*.

---

**Example 1 - Attempting a connection using a port number** In this example, the `model_ready` function attempts a connection to a server model. Because the client and server models are running on the same machine, the `MachineName` function is used to obtain the name of the server machine (see "MachineName function" on [page C.6](#)). The port number that is used for the socket connection can be any number greater than 1000; in this case, it is port 3554.

```
begin model_ready function
  set V_server to MachineName()
  /* Vserver is a string variable */
  set V_sock1 to ConnectSocketPort(Vserver, 3554)
  /* V_sock1 is a variable of type SocketPtr;
     V_server is a string variable set to
     the name of the server machine;
     3554 is the port number. */
  return true /* return value not used */
end
```

**Example 2 - Attempting a connection using a service name** In this example, the `model_ready` function attempts a connection to a server machine named `host.example.com`. Before you can attempt a connection using a service name, you must define the service name (see "Defining a service name" on [page 3.6](#)).

```

begin model ready function
  set V_sock1 to ConnectSocketService("host.example.com",
    "model1")
  /* V_sock1 is a variable of type SocketPtr;
     host.example.com is the name of the server machine;
     model1 is the service name. */
  return true /* return value not used */
end

```

## Accepting a connection (server)

There are two AutoMod functions that you can use to send a connection acceptance to a client. Choose the function based on whether you want to open the connection using a service name or port number (see “Defining a port for a socket connection” on [page 3.6](#)).

**To accept a connection using a:** Set a variable of type SocketPtr using this function:

|              |                                 |
|--------------|---------------------------------|
| Port number  | AcceptSocketPort(<Port_number>) |
| Service name | AcceptSocketService(<Service>)  |

The parameter of each function identifies the port number (an integer) or service name (a string) to use when accepting the connection. For more information, see appendix C, “Socket Communication Functions.”

The return value of both functions is a socket pointer that points to the open connection; this pointer is used later to send and read messages using this socket connection.

### Tip

- If you want to accept connections at the beginning of a simulation, call the acceptance functions from within the `model ready` function. For more information, see “Time-specific functions” in the “Process system” chapter of the *AutoMod User’s Guide*.
- If you want to continuously accept connections throughout a simulation, you can use the `SetSocketMessages` function to turn off timeout warning messages when a connection is not established (see “Example 3 – Continuously accepting connections” on [page 3.11](#)).

**Example 1 – Accepting a connection using a port number** In this example, the `model ready` function accepts a connection using a port number. The port number that is used for the socket connection can be any number greater than 1000; in this case, it is port 3554.

```

begin model ready function
  set V_sock1 to AcceptSocketPort(3554)
  /* V_sock1 is a variable of type SocketPtr */
  return true /* return value not used */
end

```



**Example 2 - Accepting a connection using a service name** In this example, the `model_ready` function accepts a connection using a service name. Before you can use a service name to accept a connection, you must define the service name (see “Defining a service name” on [page 3.6](#)).

```
begin model_ready function
  set V_sock1 to AcceptSocketService("model1")
  /* V_sock1 is a variable of type SocketPtr;
     model1 is the service name. */
  return true /* return value not used */
end
```

**Example 3 - Continuously accepting connections** In this example, the `Faccept` function is defined to check for incoming client connections. The `Faccept` function is called from within the `model_nextclock` function so that the model accepts connections continuously throughout the simulation (for more information, see “model\_nextclock function” on [page C.19](#)).

The server is set to time out immediately if a client is not attempting to connect. The `SetSocketMessages` function is used to prevent a warning message each time the server times out. For more information, see “SetSocketMessages function” on [page C.16](#).

```
begin Faccept function
  call SetAcceptBlocking(0) /* set acceptance to timeout */
  call SetAcceptTimeOut(0) /* set timeout length to zero */
  call SetSocketMessages(0) /* turn off messages */
  set V_clientcount to 1
  while V_clientcount <= V_totalclients do begin
    if ValidSocket(V_clientsocket(V_clientcount)) = false then begin
      set V_clientsocket(V_clientcount) to AcceptSocketPort(Vportnum)
      if ValidSocket(V_clientsocket(V_clientcount)) = true then begin
        call F_Action
      end
    end
    inc V_clientcount by 1
  end
  return true /* return value not used */
end
```

## Verifying a connection

There are several ways to get information about a socket connection:

- Printing socket pointer information
- Getting socket information using functions
- Calling a function to verify that a socket connection is valid

### Printing socket pointer information

You can print a socket pointer expression, which consists of the socket’s number (a random identification number assigned to the socket) followed by the name of the machine to which the socket is connected, for example, `Socket 164 to host.example.com`.

**Example 1 – Printing socket information** Printing socket information is useful when a model is accepting connections from several client machines. In this example, the `model_ready` function is defined to accept connections from three clients. An arrayed variable, `V_clientsocket`, is used to point to the open socket connections.

```
begin model_ready function
  set V_totalclients to 3
  set V_portnum to 9999

  call SetNumAcceptsPort(V_portnum, V_totalclients)
  /* allow multiple connections using same port number */
  set V_clientcount to 1
  while V_clientcount <= V_totalclients do begin
    set V_clientsocket(V_clientcount) to AcceptSocketPort(V_portnum)
    print "Connected: " V_clientsocket(V_clientcount) to message
    inc V_clientcount by 1
  end
  return true /* return value not used */
end
```

Each time a connection is opened, the socket pointer is printed to the Messages dialog box to identify the name of the connected client machine.

## Getting socket information using functions

You can use the `GetSocketName` function to return the same information that is obtained by printing a socket pointer (see “`GetSocketName` function” on [page C.5](#)). For example, the following code sets `V_string` equal to the socket number (a random identification number assigned to the socket) followed by the name of the machine to which the socket is connected:

```
V_string = GetSocketName(Vsock1)
/* Vstring is a string variable;
   Vsock1 is a variable of type SocketPtr. */
```

There are also functions for returning just the socket number, or just the connected machine name. See “`GetSocketNum` function” on [page C.6](#) and “`GetSocketHost` function” on [page C.5](#).

## Verifying that a socket connection is valid

The `ValidSocket` function allows you to verify that a socket connection is open and capable of sending and receiving messages. In this example, a `while` loop is defined to continuously read from a socket until a message is received (the function used to read a message is described later in this chapter). The `ValidSocket` function is used to prevent a continuous loop if the connection closes and a message cannot be read.

```
while (ValidSocket(V_syncsock) = true and V_msg = null) do begin
  set V_msg = ReadSocketString(V_syncsock)
  /* ReadSocketString function described later in this chapter */
end
```

If the connection becomes invalid for any reason, the model stops trying to read a message.

## Closing a connection

All socket connections close automatically at the end of simulation and when AutoMod is closed. Connections are also automatically closed if there is an error when reading or sending a message. If you want to manually close a socket connection during a simulation, you can call the `CloseSocket` function in any function or procedure (see “CloseSocket function” on [page C.3](#)).

---

## Sending and reading messages

After opening one or more connections, you can call functions to send and read messages from a client to a server or vice versa. A *message* is any data sent through a socket connection. The format of a message determines the type of data that the message can contain (for example, string, real, integer, and so on).

## Message formats

Messages that are sent and read through a socket connection can be one of the following formats:

- String
- C structure

---

**Note** A different format is used for communicating between models that are synchronized by the MMS Server. For more information, see “Communicating between synchronized models” on [page 4.7](#).

---

### String messages

A *string* message is composed of one or more alphanumeric characters. `Create 2 small` is an example of a possible string message. String messages contain only one type of data (type `String`) and can be sent and read in a model using standard AutoMod code.

### C structure messages

A *C structure* message is capable of storing many different types of data (for example, `String`, `Integer`, `Short`, and so on). To send a C structure message, you must first define a structure in C code. A *structure* is a collection of related variables that are stored in one name. Each variable in a structure is called a *member*.

For example, you could define a structure for the message `create 2 small`, as follows:

```
struct gen{
    char vact[10]; /* holds the string "create" */
    int vquant; /* holds the integer "2" */
    char vltype[10]; /* holds the string "small" */
};
```

The structure `struct gen` has three members: an array of 10 characters, an integer value, and another array of 10 characters. C structures are not supported directly in AutoMod code; C structure messages can only be sent and read from custom C functions called within a model.

! **Important** Structures must be defined the same way in both the sending and receiving systems.

When transferring C structure messages containing numeric types between a model and a control system on a platform that uses the network byte order, you need to convert the numeric values to the network standard.

**Converting numeric values in C structure messages to the network byte order** The order in which the bytes of a multi-byte number are stored in memory on a Windows machine is least-significant byte first (also known as “little-endian” storage). The order in which the bytes are transmitted on a network and stored in memory on most Unix platforms is most-significant byte first (“big-endian” storage). When sending numeric values to a control system running on a platform that uses the network byte order, you must first convert the numeric values to the network byte order. Likewise, if your model receives numeric values from a control system running on a platform that uses the network byte order, you must convert the numeric values back to the Windows sorting order before using the values in the model.

---

**Note** If you are only transferring messages between models, you do not need to convert numeric values to the network standard. Numeric values in a model must be converted to the network standard only if you are transferring C structure messages containing numeric types (for example, Integer, Real, and so on) through a socket connection with a machine that stores values using the network byte order.

---

There are four functions that can be used to convert numeric values. Choose the function based on the numeric type of the value that you are reading or sending in a C structure message:

| Numeric types |  | Use this conversion function:             |
|---------------|--|---|
| C types       | AutoMod types                                      |   |
| Short         | N/A  | <code>NetworkConvertShort(Value)</code>   |
| Integer, Long | Integer  | <code>NetworkConvertInteger(Value)</code> |
| Float         | N/A  | <code>NetworkConvertFloat(Value)</code>   |
| Double        | Real, Time,<br>Distance, Velocity,<br>Acceleration | <code>NetworkConvertDouble(Value)</code>  |

**Example - Converting numeric values** In this example, a message containing integer and time values is being sent from a model to a control system on a Unix machine. The conversion functions used when sending these data types are `NetworkConvertInteger` and `NetworkConvertDouble` respectively (see the table above).

```
struct example{ /* structure definition */
    int Vint;
    double Vtime;
};

struct example vmsg;

vmsg.Vint = NetworkConvertInteger(12);
/* integer value 12 is converted to network standard */
vmsg.Vtime = NetworkConvertDouble(Vsecs);
/* time value in vsecs converted to network standard */
...
```

The variables `Vint` and `Vtime` are set to the converted numeric values. These values are stored in the variable `vmsg`, of type `struct example`.

---

**Note** If the control system sends any messages back to the model, received numeric data values must also be converted before the values can be used in the model.

---

## Sending a message

Models send a message by calling a function in AutoMod or C code. The function used for sending a message depends on the message format:

To send a message of this type:

Use this function:

|             |  |
|-------------|--|
| String      | <code>SendSocketString(&lt;Socket&gt;, &lt;Message&gt;)</code>                   |
| C structure | <code>SendSocket(&lt;Socket&gt;, &lt;Msg_structure&gt;, &lt;Msg_size&gt;)</code> |

When you call a function to send a message, the message is immediately sent through the specified socket connection and travels to its destination, where it waits to be read (see “Reading a message” on [page 3.19](#)).

**Tip** You must know the format of a sent message in order to read it. If you are sending both string and C structure messages, open a separate socket connection for each message format; you can then keep track of which function to call when reading messages from a specific socket.

### Sending a string message

String messages are sent by passing a socket pointer and string expression to the `SendSocketString` function. The socket pointer points to the connection through which you want to send a message. The string expression defines the message content. For more information, see “SendSocketString function” on [page C.11](#).

**Example - Sending a string message** Suppose you are using the Model Communications module to simulate loads transferring from model A to model B. The easiest way to model the transfer is to send loads to die in model A and generate new loads of the same type in model B at the same simulation time.

Before a load is sent to die in model A, it calls a function to send a message to model B containing an action, a quantity, and a load type. Model B uses this information to generate new loads of the same type.

An AutoMod function that sends the string message can be defined as follows:

```
begin F_CustomSendString
  set V_msg to "create 1 large" /* V_msg is of type String */
  set V_size to SendSocketString(V_sockString, V_msg)
  /* V_size is of type Integer;
     V_sockString is of type SocketPtr */
  if V_size = 0 then begin
    /* connection closes automatically due to send error */
    print "Warning! Message send failed" to message
    set V_sockString to null
```

```

        return 0
    end
    return 1
end

```

The parameters of the `SendSocketString` function define the socket through which the message is sent (the socket pointed to by the variable `V_sockString`), and the content of the message (the character data stored in the string variable `V_msg`).

The integer variable `V_size` is set to the return value of the `SendSocketString` function (the number of bytes sent). If the variable is equal to zero (indicating that the send could not be completed), then a warning is displayed in the Messages dialog box.

---

**Note** The connection closes automatically if an error occurs while sending the message, but the variable `V_sockString` must be manually set to null (because it is no longer pointing to an open socket connection).

---

## Sending a C structure message

C structure messages are sent by passing a socket pointer, string, and integer expression to the `SendSocket` function. The socket pointer points to the connection through which you want to send a message. The string expression defines the message structure (values in the C structure are converted to a string) and the integer expression defines the size of the message in bytes. For more information, see “SendSocket function” on [page C.10](#).

**Example - Sending a C structure message** This example demonstrates how you can send C structure messages to simulate loads transferring from one model to another. Using a structure, you can define each part of the message in a variable of a different type. In this example, the message contains an action (a string), a quantity of loads (an integer), and a load type (another string).

The user-defined AutoMod function `F_sendCmessage` passes the action, load quantity, and load type to the user-defined C function

`CustomSendSocketCMsg:`

```

begin F_sendCmessage
    set V_csize to CustomSendSocketCMsg(V_sockCstruct, "create",
    1, "large")
    /* C function is passed socket pointer, string, integer,
    and string */
    if V_csize = 0 then begin
        /* connection closes automatically due to send error */
        print "Warning! Message send failed" to message
        set V_sockCstruct to null
        return 0
    end
    return 1
end

```

The function sets the integer variable `v_csize` to the return value of the C function (the number of bytes sent). If the variable is equal to zero (indicating that the send could not be completed), then a warning is displayed in the Messages dialog box.

---

**Note** The connection closes automatically if an error occurs while sending the message, but the variable `v_sockCstruct` must be manually set to null (because it is no longer pointing to an open socket connection).

---

The user-defined C function `CustomSendSocketCMsg` copies the message content into a structure and sends the C structure message:

```
/* sending a C structure message */

#include "user.h"

struct gen{ /* structure definition */
    char Vact[10];
    int Vquant;
    char Vltype[10];
};

int
CustomSendSocketCMsg(amSocket* vsocketptr, char *act, int
quant, char *ltype)
{
    struct gen Vmsg;
    /* defines a structure variable for message content */
    int Vret; /* defines an integer for return value */
    strcpy(Vmsg.Vact, act);
    /* defines the action in the structure message */
    Vmsg.Vquant = quant;
    /* defines the load quantity */
    strcpy(Vmsg.Vltype, ltype);
    /* defines the load type */
    Vret = SendSocket(vsocketptr, (char*)&Vmsg,
sizeof(struct gen));
    /* SendSocket function sends C structure message */
    return Vret;
    /* function returns length of sent message in bytes */
}
```

After including a header file, the function defines the structure `struct gen` to describe the data types contained in the message. The structure contains three members: an array of 10 characters, an integer value, and another array of 10 characters.

❗ **Important** Structures must be defined the same in both the sending and receiving systems.



The variable `Vmsg` defines the message content; `Vmsg` is of type `struct gen` and stores the values of the structure members that are passed to the `CustomSendSocketCmsg` function (see the `FsendCmessage` function on the previous page). For example, the value of the member `Vact` is `create`, the value of the member `Vquant` is `1`, and the value of the member `Vltype` is `large`, all of which are stored in the variable `Vmsg`.

The first parameter of the `SendSocket` function points to the socket connection through which the message is sent. The second parameter is the message content converted to a string. The third parameter is the length of the message in bytes.

The `CustomSendSocketCmsg` function returns the value of the variable `Vret`, which is the number of bytes sent through the socket connection.

## Reading a message

Models read a message by calling a function in AutoMod or C code. After a message has been read, you can interpret the message and perform actions in a procedure or function based on the message content (for more information, see “Interpreting a message” on [page 3.23](#)).

The function used for reading a message depends on the message format:

| To read a message of this type: | Use this function:   |
|---------------------------------|--|
| String                          | <code>ReadSocketString&lt;Socket&gt;</code>                                      |
| C structure                     | <code>ReadSocket(&lt;Socket&gt;, &lt;Msg_structure&gt;, &lt;Msg_size&gt;)</code> |

Messages that are sent through a socket connection accumulate at their destination and wait to be read. Waiting messages can be read using the `ReadSocketString` or `ReadSocket` function. You can set the type of socket connection to define how long these functions attempt to read a message before giving up (see “Defining the length of reading attempts” on [page 3.20](#)). If multiple messages arrive, they are not overwritten; to read all waiting messages, you can use a while loop to continuously read messages from a timeout socket, as shown in the examples in this section.

**Tip** You must know the format of a sent message in order to read it. If you are receiving both string and C structure messages, open a separate socket connection for each message format; you can then keep track of which function to call when reading messages from a specific socket.

## Defining the length of reading attempts

String and C structure messages that are sent through a socket connection wait at their destination until they are read. If a message is waiting when a function reads from the socket connection, the waiting message is read immediately. If there is not a message waiting at the socket, the function continues attempting to read a message. The length of time the function tries to read a message depends on whether the socket is set to blocking or timeout.

If a socket is set to *blocking*, it means that a function attempting to read a message at that socket delays the simulation indefinitely until a message is read.

---

**Note** A socket that is set to blocking disables keyboard and mouse input and causes the simulation to delay until a message is read.

---

If a socket is set to *timeout*, it means that a function attempting to read a message at that socket waits for a specified length of time; if a sent message does not arrive within the defined time period, the function stops trying to read a message and must be called again.

By default, sockets are set to time out in zero seconds (if a message is not waiting at the exact time the function tries to read a message, then the function stops trying and no message is read). You can set sockets to blocking or timeout by calling the `SetSocketNonBlocking` function. You can set the read length of a timeout socket by calling the `SetSocketTimeOut` function in any procedure or function; for more information, see appendix C, “Socket Communication Functions.” The length of reading attempts at a socket can be changed at any time during simulation.

**Example - Setting a blocking socket** In this example, the `model_ready` function is used to set socket `V_sockB` to blocking.

```
begin model_ready function
  call SetSocketNonBlocking(V_sockB, 0)
  /* V_sockB is a variable of type SocketPtr;
     0 is a flag to turn on blocking. */
  return true /* return value not used */
end
```

The second parameter (the zero integer) indicates that the socket is blocking. A function that attempts to read a message from `V_sockB` delays the simulation until a message is read.

## Reading a string message

String messages are read by passing a socket pointer to the `ReadSocketString` function. The socket pointer points to the connection from which you want to read a message. For more information, see “`ReadSocketString` function” on [page C.9](#).

**Example - Reading a string message** In this example, a custom AutoMod function (`F_readallStrings`) reads a string message from the specified socket and calls another custom function (`F_interpretStrings`) to interpret the message. The function then reads from the socket again to see if another message is waiting. As long as messages arrive (`V_msg <> null`) and the connection remains open (`ValidSocket(V_sockString) = true`) the function continues to read from the socket.

**Important** If you are using a while loop to continuously read messages from a socket, the socket must be set to time out (see “Defining the length of reading attempts” on [page 3.20](#)). Otherwise, the model will try to read a message, even after the last message has been read, resulting in an infinite loop.

```
begin F_readallStrings function
  set C_read to 0 /* initialize a counter */
  if ValidSocket(V_sockString) = false then return 0
  /* if the connection is closed, do not continue */
  set V_msg to ReadSocketString(V_sockString)
  /* Vmsg is a variable of type String */
  while (ValidSocket(V_sockString) = true and V_msg <> null)
  do begin
    increment C_read by 1 /* count the messages */
    call F_interpretStrings(V_msg)
    /* Finterpret is a custom function that performs
       actions based on the message content */
    set V_msg to ReadSocketString(V_sockString)
    /* check for another message */
  end
  return C_read /* return the number of read messages */
end
```

The `ReadSocketString` function reads the message data and stores it in the variable `V_msg`. As soon as the message arrives, the `F_interpretStrings` function is called to perform actions based on the message content (see “Interpreting a message” on [page 3.23](#)). The `ReadSocketString` function then checks the socket to see whether another message is waiting.

## Reading a C structure message

C structure messages are read by passing a socket pointer, string and integer expression to the `ReadSocket` function. The socket pointer points to the connection from which you want to read a message. The string expression defines the message structure (values from the received message are converted and stored in the structure) and the integer expression defines the size of the message in bytes. For more information, see “ReadSocket function” on [page C.9](#).

**Example - Reading a C structure message** In this example, a custom AutoMod function (`F_readallCStructs`) calls a custom C function to read and interpret a C structure message waiting at the specified

socket. After the message is read, the C function is called again to see if another message is waiting at the socket. As long as messages are arriving (`V_msg <> null`) and the connection remains open (`ValidSocket(V_sockCstruct) = true`) the function continues to read from the socket.

**Important** If you are using a while loop to continuously read messages from a socket, the socket must be set time out (see “Defining the length of reading attempts” on [page 3.20](#)). Otherwise, the model will try to read a message, even after the last message has been read, resulting in an infinite loop.

The custom AutoMod function is defined as follows:

```
begin F_readallCstructs function
  set C_read to 0 /* initialize a counter */
  if ValidSocket(V_sockCstruct) = false then return 0
  /* if the connection is closed do not continue */
  set V_msgread to CustomReadSocketCMsgs(V_sockCstruct)
  /* V_msgread is a string variable;
     V_sockCstruct is a socket pointer. */
  while (ValidSocket(V_sockCstruct) = true and V_msg <> null)
    do begin
      increment C_read by 1 /* count the messages */
      set V_msgread to CustomReadSocketCMsgs(V_sockCstruct)
      /* keep reading until all messages have been read */
    end
  return C_read /* return the number of read messages */
end
```

The string variable `V_msgread` is set to the return value of the C function, indicating whether or not a message was read successfully (if the value is `null`, the read failed).

The source file containing the C function used to read messages is defined as follows:

```
/* reading a C structure message */
#include "user.h"

struct gen{ /* structure definition */
  char Vact[10];
  int Vquant;
  char VltypeStr[10];
};

char *
CustomReadSocketCMsgs(amSocket* Vsocketptr)
{
  struct gen Vmsg;
  /* define a structure variable for the message content */
  int Vret; /* define an integer for the return value */
  Vret = ReadSocket(Vsocketptr, (char*)&Vmsg,
    sizeof(struct gen));
}
```

```

        /* ReadSocket function is a standard function used to
        read a C structure message */
    if (Vret != 0) { /* if a message was read */
        FinterpretCStructs(Vmsg.Vact, Vmsg.Vquant,
            Vmsg.VltypeStr);
        /* call AutoMod function to interpret message;
        pass values stored in message structure. */
        return "Read successful";
        /* return string to indicate message was read */
    }
    else {
        return null;
    }
}

```

The structure `struct gen` is defined to describe the data types contained in the message. The structure contains three members: an array of 10 characters, an integer value, and another array of 10 characters.

**! Important** Structures must be defined the same in both the sending and receiving systems.

The variable `Vmsg` is of type `struct gen` and stores the content of the read message. Once stored in the structure variable, the values are passed to the `AutoMod` function `F_interpretCStructs` which performs actions based on the message content (see “Interpreting a message” on [page 3.23](#)). The function then returns a string to indicate that a message was read successfully.

## Interpreting a message

*Interpreting* a message is the process of writing user-defined functions or procedures that define which actions should be performed based on the message content. Messages consist of one or more characters that are either all of type `String` (a string message) or are of various data types (a C structure message). A message from a control system might be a string of numbers, while a message from a model might contain alphabetic characters. Regardless of the type of data contained in a message, the content of a message is meaningless to the recipient model until the message is interpreted.

This section includes examples that show how to:

- Interpret a string message.
- Interpret a C structure message.

## Example 1 - Interpreting a string message

In this example, the content of a string message consists of an action, a quantity, and a load type (see “Example – Sending a string message” on [page 3.16](#)). Once the model has read the message, you can call a user-defined function to create loads according to the message specification:

```
begin F_interpretCstrings function
  if (ArgMsg <> null) then begin
    read V_act, V_quant, V_ltype from ArgMsg
    /* V_act is a variable of type String;
       V_quant is a variable of type Integer;
       V_ltype is a variable of type LoadTypePtr. */
    if (V_act = "create") then begin
      if (V_quant <> 0) then begin
        if (V_ltype <> null) then begin
          create V_quant loads of load type V_ltype to
            P_start
        end
      end
    end
    if (V_act = "order") then begin
      if (V_quant <> 0) then begin
        /* load type is ignored */
        order V_quant loads from OL_supply to continue
      end
    end
    print "Warning! Message ignored" to message
    return 0
  end
end
```

The `read` action uses the default whitespace delimiter to read the content of the message. The `read` action automatically casts the load quantity to an integer (stored in the variable `V_quant`) and the load type to a `LoadTypePtr` (stored in the variable `V_ltype`).

The function uses the first string value read from the message to determine whether to create or order loads in the system. If the first string in the message is `create`, the function uses the next two values read from the message to determine the number and type of loads to create. The function then sends newly created loads to the process `P_start`.

If the first string in the message is `order`, the function orders the specified number of loads from the order list `OL_supply` to continue (the load type is ignored).

If the first string in the message is not an anticipated string, the function prints a warning to the Messages dialog box.

## Example 2 - Interpreting a C structure message

In this example, the content of a sent message consists of an action, a quantity, and a load type (see “Example – Sending a C structure message” on [page 3.17](#)). Once the message has been read, you can call a user-defined function to create loads according to the message specification:

```
begin F_interpretCStructs function
  set V_ltype to ArgLtypeStr
  /* converts the string expression to a load type */
  if ArgAct = "create" then begin
    create ArgQuant loads of load type V_ltype to P_start
    return 1
  end
  if ArgAct = "order" then begin
    order ArgQuant loads from OL_supply to continue
    in case order not filled backorder on OL_back
    return 1
  end
  print "warning! Message ignored" to message
  return 0
end
```

`F_interpretCStructs` has three parameters of type String, Integer, and String respectively. These parameters define an action, a quantity of loads, and a load type that are used to interpret the message.

First, the load type variable `V_ltype` is set to the parameter `ArgLtypeStr`, which converts the message’s string expression to an AutoMod load type.

If the action is `create`, the function uses the `ArgQuant` parameter and the `V_ltype` variable to determine the number of loads to create and the load type.

If the action is `order`, the function uses the `ArgQuant` parameter to order a number of loads from an order list to continue; in this case, the load type is ignored.

If the action is not an anticipated string, the function prints a warning to the Messages dialog box.

---

## Synchronizing two models without the MMS Server

If you have purchased the basic Model Communications module and an additional runtime license of AutoMod, you can synchronize two models by writing functions that alternately cause one model to delay while allowing the other model to complete its current events. For more information, see “Exact synchronization” on [page 4.4](#) (periodic synchronization is not available without the MMS Server).

**Important** Synchronizing two models without the MMS Server is not recommended. Without the server, the number of models that you can synchronize is limited to two, and you are required to write code to open socket connections and check the time of future events in each model.

When using the Model Communications module to synchronize models, a model’s graphics and simulation clock may temporarily drop out of synch if you turn off the animation or pause one of the synchronized models during simulation. The model temporarily delays because the animation display and the clock are not updated until the simulation advances to the time of the next future event. As soon as the events on the delayed model’s future event list are moved to the current event list, the model graphics and simulation clocks are synchronized.

The code used for synchronizing two models without the MMS Server is provided in the following example.

### Example - Synchronizing two models without the MMS Server

This example demonstrates how to synchronize two models without using the MMS Server. The synchronization code is the same in both the client and server models. The `model_ready` function is used to open two socket connections: one for model data and the other for synchronization messages.

```
begin model_ready function
  if (V_servermodel = 1) then begin /* server model */
    set V_active to true
    /* V_active is an integer variable that indicates
       the active model during simulation. Initially, the
       server is the active model. */
    call SetAcceptBlocking(1)
    call SetNumAcceptsPort(5555, 2)
    set V_syncsock = AcceptSocketPort(5555)
    set V_datasock = AcceptSocketPort(5555)
  end
```



```

else begin /* client model */
    set V_active = false
    set V_servername = "host.example.com"
    /* the string variable V_servername is set to the
       name of the server machine */
    call SetConnectBlocking(1)
    set V_syncsock = ConnectSocketPort(V_servername, 5555)
    set V_datasock = ConnectSocketPort(V_servername, 5555)
end
return true /* return value not used */
end

```

---

**Note** The initial value of the integer variable `V_servermodel` is set to 1 on the server machine and 0 on the client machine.

---

The socket connection pointed to by `V_syncsock` is used for sending and receiving synchronization messages while the connection pointed to by `V_datasock` is used to send and receive data messages between models.

After the models are initialized, the `model nextclock` function is used in both models to send and receive messages (see “model nextclock function” on [page C.19](#)).

---

**Tip** The `model nextclock` function is the ideal place for sending and receiving synchronization messages because it is called after all current events have been completed.

---

```

begin model nextclock function
    call F_readallStrings /* read model data messages */
    call F_syncmodels /* synchronize models */
    return true /* return value not used */
end

```

In this example, two user-defined functions are called to send and receive messages. `F_readallStrings` is a function that is called to read and interpret model data messages; when all received messages have been interpreted, the `F_syncmodels` function is called to send and receive synchronization messages.

The `F_syncmodels` function, which sends and reads synchronization messages, is defined as follows:

```
begin F_syncmodels function
  if (ValidSocket(V_syncsock) = false) then
    /* if there is no valid socket connection */
    return 0
  set V_nextclock to FEvFirstTime()
    /* set the variable V_nextclock to the time of the next
      future event */
  print V_nextclock as .6 to V_msg
    /* reduce the precision of the time variable */
  set V_nextclock to V_msg
  if (V_active = true) and (V_nextclock > V_otherclock)
    then begin
      /* if the other model has the earliest future event */
      set V_active to false /* this model delays */
      print V_nextclock as .6 to V_msg
      call SendSocketString(V_syncsock, V_msg)
        /* send time of this model's next future event to the
          other model */
    end
  while (V_active = false) and (ValidSocket(V_syncsock) =
    true) do begin
    /* while this model is delaying */
    set V_msg = ReadSocketString(V_syncsock)
    /* try to read a time message */
    while (V_msg = null and ValidSocket(V_syncsock) = true)
      do begin
        /* create a loop to delay the model until a time
          message is received */
        set V_msg = ReadSocketString(V_syncsock)
      end
    set V_otherclock to V_msg
    if V_otherclock >= V_nextclock then begin
      /*if this model has the earliest future event */
      set V_active to true
    end
  end
  return true /* return value not used */
end
```

When the `F_syncmodels` function is called, both models use the `FEvFirstTime` function to determine the time of their next future event (see “`FEvFirstTime` function” on [page C.5](#)). The time value is reduced to the sixth decimal place and sent in a message between models. Both models compare the time of the next future event, and the model with the earliest future event continues processing.

**Important** Any action that causes a delay can change the time of the next future event. Do not use actions that cause a delay after calling the `FEvFirstTime` function and before checking time messages to see which model has the earliest future event.

The model that does not have the earliest future event resends the time of its next future event and then delays processing by continuously reading from a timeout socket connection for a time message from the active model. When the message is received, the times are again compared, and the cycle repeats.





# Multi-Model Synchronization (Model Communications Plus module)

In this chapter, you will learn:

- How events determine the rate of simulation advancement
- The difference between exact and periodic synchronization
- How to communicate between synchronized models
- How to start a synchronized simulation
- How to use trace files

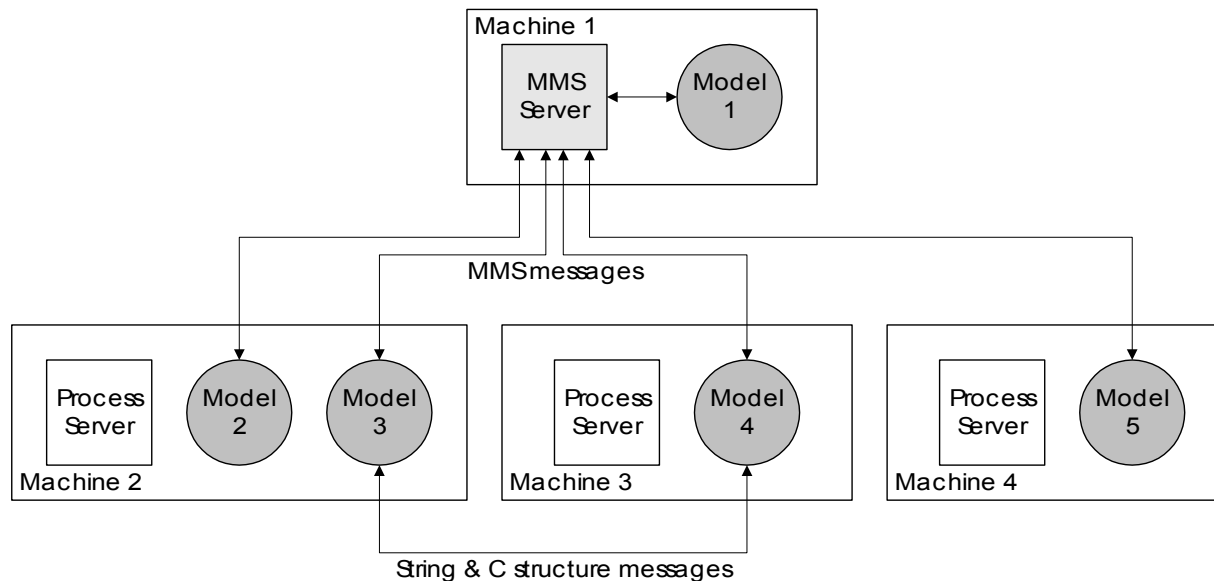
---

## Introduction

*Synchronization* is the process of making the simulation clock advance in multiple models at the same rate. Synchronization is necessary if you are running interactive models that communicate to schedule future events. For example, if model A sends a message to model B to create a load at time 30 in the simulation, it is important that model B not advance past time 30 before it receives the message. As long as both models are synchronized, they can coordinate and schedule simulation events.

If you have purchased the Model Communications Plus module, you can use the Multi-Model Synchronization Server (MMS Server) to synchronize two or more models; the number of models you can synchronize is limited only by the number of runtime licenses you own. The MMS server automatically opens and manages a connection between each model and the server and synchronizes all of the models that are participating in the simulation.

The following diagram shows one possible configuration of models that are synchronized during a simulation:



The synchronized models communicate by sending MMS messages to the server. When the MMS Server receives a message from a model, it broadcasts the message either to all of the synchronized models in the simulation (the default) or to one specific model. Each model that receives a message can then read the message and perform actions based on the message content, or ignore the message if it does not apply to that model. The messages that are used to communicate between models and the MMS Server can contain any combination of string, real, and integer values. MMS messages are discussed later in this chapter.

---

**Note** You can also open connections directly between synchronized models (see Model 3 and Model 4 in the preceding diagram) and communicate using string or C structure messages as described in chapter 1, “Socket Communications.”

---

In order to understand how models are synchronized, it is first necessary to understand how model events determine the rate of simulation advancement.

## Understanding event-driven simulation

AutoMod models advance event by event through time. When all events for a time (such as 10.06 seconds) are completed, the simulation clock advances to the next future event. An *event* is any

action performed by an entity during a simulation (for example, a vehicle claiming a control point, a load releasing an operator, and so on).

---

**Note** Display steps that update animation and statistics are events; if your display step is set to one second (the default), then there is at least one event per second.

---

In AutoMod, events are sorted on two lists:

- The current event list
- The future event list

The *current event list* is a list of the events that are waiting to be performed at the current simulation clock time (for example, 3:18:24.03). Events are assigned a priority by the software, and are sorted with the highest-priority events listed first. Events are removed from the current event list as soon as they are performed. When all of the events have been removed from the current event list, the simulation clock advances to the time of the next event on the future event list, and the events scheduled for that time become the current event list.

The *future event list* is a list of events that are scheduled to be performed at a future time. Events are sorted by simulation clock time with the lowest value (earliest time) first. When the current simulation time advances, the events scheduled at the earliest simulation time are moved to the current event list.

## Asynchronous simulation clocks

When models are not synchronized, they advance simultaneously according to the events on each model's event lists. As soon as a model's current events are completed, the model's simulation clock advances to the time of the next event for that model. The following table illustrates how two *asynchronous* models advance during simulation.

| Model 1 |   | Model 2 |   |
|---------|---|---------|---|
| Time    | Events                                    | Time    | Events  |
| 100     | Load generated.<br>Load moves into queue. | 100     | Load generated.<br>Load moves into control point.   |
| 110     | Resource claims load.                     | 105     | Vehicle picks up load.<br>Vehicle claims destination control point.<br>Vehicle begins moving. |
| 115     | Resource frees load.                      | 118     | Vehicle arrives at destination.<br>Vehicle sets down load.                                    |
| 125     | Load moves into station.                  | 125     | Load moves into queue.  |

The simulation clocks advance simultaneously in different increments; for example, when the current events at time 100 are completed, model 2 advances to time 105, while model 1 skips time 105 (because there are no events) and advances to time 110.

## Synchronizing simulation clocks

Simulation clocks are synchronized by delaying one or more model's events to make the models advance at the same rate during a simulation.

---

**Tip** You must set the display step in all models to the same interval in order to synchronize the animation during a simulation.

---

There are two methods of model synchronization:

- Exact synchronization
- Periodic synchronization

When starting the MMS Server, you can use the **-me** or **-mp** command line options to define exact or periodic synchronization, respectively (for more information, see “Command line options” on [page 4.23](#)). The method of synchronization determines how frequently models are synchronized. By default, the MMS Server synchronizes models exactly.

### Exact synchronization

When models are exactly synchronized, they do not process events at the same time; one model is active while the other models delay. An *active model* is a synchronized model that advances to the time of its next future event and completes the events on its current event list. Although only one model is active at a time, events that are scheduled at the same time in multiple models appear to be performed simultaneously during the simulation (depending on the speed of your machine's processor).

---

**Tip** Exact synchronization should be used when many model events occur within short periods of simulation time.

---

Any model can start the simulation as the active model. After the active model completes its current events, the time of its next future event is compared with that of the delayed models. The model with the earliest future event becomes the active model and advances its simulation clock. When the active model's current events are completed, the time of the next future event is compared with the delayed models' next future events, and the cycle repeats.



The following table illustrates how two exactly synchronized models advance during simulation.

| Model 1 |  | Model 2 |   |
|---------|--|---------|---|
| Time    | Events   | Time    | Events  |
| 100     | Load generated.<br>Load moves into queue.<br>Compare time of next future event.<br>Delaying. | 100     | Delaying.<br><br>Load generated.<br>Load moves into control point.<br>Compare time of next future event.                            |
|         |  | 105     | Vehicle picks up load.<br>Vehicle claims destination control point.<br>Vehicle begins moving.<br>Compare time of next future event. |
| 110     | Resource claims load.<br>Compare time of next future event.                                  |         | Delaying.   |
| 115     | Resource frees load<br>Compare time of next future event.<br>Delaying.                       |         |   |
|         |  | 118     | Vehicle arrives at destination.<br>Vehicle sets down load.<br>Compare time of next future event.                                    |
| 125     | Delaying.<br><br>Load moves into station.<br>Compare time of next future event.              | 125     | Load moves into queue.<br>Compare time of next future event.<br>Delaying.   |

At the beginning of the simulation, model 1 is the active model. When model 1 completes its current events, the model with the earliest future event (model 2 at time 100) becomes the active model. Throughout the simulation, the model with the earliest future event becomes the active model and completes its current events while the other model delays.

**Note** When events occur at the same time in multiple models (for example, time 125 above), the models take turns completing their events for that simulation time (it does not matter which model completes its events first).

## Periodic synchronization

Periodically synchronized models process events simultaneously. When a model completes all of the events that have been scheduled for a synchronization period (for example, 10 seconds), the model delays until the other models in the simulation have completed their events for the same time interval. All models then advance their simulation clocks to the next time interval and begin again.

For example, the following table illustrates how two models are periodically synchronized at 10 second intervals by the MMS Server:

| Model 1 |  | Model 2 |   |
|---------|--|---------|---|
| Time    | Events   | Time    | Events  |
| 100     | Load generated.<br>Load moves into queue.<br>Delaying. | 100     | Load generated.<br>Load moves into control point.   |
|         |  | 105     | Vehicle picks up load.<br>Vehicle claims destination control point.<br>Vehicle begins moving. |
| 110     | Synchronization update<br>Resource claims load.        | 110     | Synchronization update  |
| 115     | Resource frees load                                    | 118     | Vehicle arrives at destination.<br>Vehicle sets down load.                                    |
| 120     | Synchronization update                                 | 120     | Synchronization update  |
| 125     | Load moves into station.                               | 125     | Load moves into queue.  |

In this simulation, model updates occur every 10 seconds (time 100, time 110, time 120, and so on). Model 1 completes its events at time 100 and then delays until model 2 completes the events it has scheduled at time 105. Once these events are completed, both models advance to time 110. Model 1 begins completing the events it has scheduled at time 110, while Model 2 simultaneously begins completing the events it has scheduled at time 118. The models' simulation clocks are synchronized only temporarily, at 10 second intervals throughout the simulation.

Periodic synchronization results in better performance than exact synchronization because models can complete events simultaneously and model delays are less frequent. The longer the synchronization period, however, the less accuracy you have when responding to received messages and scheduling future events.

When *exactly* synchronized models receive an MMS message, the `model mmsyncmessage` function is called by the receiving models at the same simulation time as when the message was sent by the originating model (see “`model mmsyncmessage` function” on [page D.18](#)). This allows you to read messages and immediately

perform actions based on the messages' content. Periodically synchronized models do not receive an MMS message at the same time the message was sent. Instead, the server sends the message at the next synchronization update and the `model mmsyncmessage` function is called at this time interval. For example, if a model sends a message at time 52.34 seconds, and the MMS Server is periodically synchronized every 15 seconds, the message is not broadcast by the server until time 60 seconds in the simulation.

---

## Communicating between synchronized models

Using the Model Communications Plus module, a simulation can be divided into several separate models that are built by different simulation engineers; each model can then later be synchronized to operate as a unified system. If you are going to build a simulation in pieces, all engineers should agree beforehand on the structure, type, and content of the messages that will be transferred during the simulation (each of these message components is discussed in greater detail later in this section). This preparation will make model communication easier when the individual models are completed.

Synchronized models communicate by sending *multi-model synchronization messages (MMS messages)* to the MMS Server. When a message is received by the server, it is broadcast to each synchronized model in the simulation (including the model from which the message originated).

MMS messages can consist of three different types of data: Integer, Real, and String. The content of an MMS message is defined by calling AutoMod functions to write values to the message for each type of data. For example, the message `68 3.5 Lhot` consists of an integer value (68), a real value (3.5), and a string value (Lhot).

MMS messages can also be assigned a numeric message type that makes message interpretation easier. For more information, see “Defining the type of an MMS message” on [page 4.10](#).

## Sending an MMS message

Before you can send an MMS message, you must call functions to set up the message in the send message buffer. The *send message buffer* is a temporary storage area in RAM where the structure, type, and content of an outgoing MMS message is defined.

### ► To set up a message in the send message buffer

1. Define the structure of MMS messages.
2. Define the content of the message.
3. Define the message's type.
4. Define the recipient of the message (optional).

After these steps are complete, you can send a message from the model to the MMS Server by calling the `MMSyncSendMessage` function. These steps are described in detail in the following sections.

### Defining the structure of MMS messages

The number of values that can be contained in outgoing MMS messages is limited by the number of fields that are defined for each type of data (for example `Integer`, `Real`, or `String`). A *field* is a space in the send message buffer that is allocated for a value of a particular type of data. The default number of fields in the send message buffer are eight integer fields, eight real fields, and eight string fields. The default length of the string fields (the number of characters that can be contained in each string) is 32 characters.

If you want to send a message containing more than eight integer values, eight real values, or eight string values of 32 characters each, you can call a function to change the structure of outgoing MMS messages. The `MMSyncSendMessageNumDatas` function allows you overwrite the default message structure and create any number of integer, real, and string fields in the send message buffer. For more information, see “`MMSyncSendMessageNumDatas` function” on [page D.11](#).

---

**Note** When you change the number of fields in the send message buffer, the new structure that you define becomes the default message structure for all subsequently sent messages.

---

As an alternative to using the `MMSyncSendMessageNumDatas` function to define the number of fields for each type of data at the same time, there are separate functions that you can use to overwrite the default number of fields of one type of data in the send message buffer. These functions are useful when you only want to change the number of fields in the buffer for one type of data:

| To define fields of this type: | Use this function:  |
|--------------------------------|---|
| Integer                        | <code>MMSyncSendMessageNumIntegers(&lt;Int_fields&gt;)</code>                         |
| Real                           | <code>MMSyncSendMessageNumReals(&lt;Real_fields&gt;)</code>                           |
| String                         | <code>MMSyncSendMessageNumStrings(&lt;String_fields&gt;,&lt;String_length&gt;)</code> |

For more information about each of these functions, see appendix D, “Multi-Model Synchronization Functions.”

---

**Tip** Not every field that you define in the send message buffer needs to contain a value when the message is sent (you can leave any number of fields empty). To avoid redefining the message structure several times during a simulation, define a message structure at the beginning of the simulation with enough fields to contain the largest message that you will send; then use the same number of fields for all messages sent throughout the simulation.

---

**Example - Defining the structure of MMS messages** This example demonstrates how to define fields in the send message buffer that can contain the following values in a message:

```
38      72      92      68      23      15      9      89      103     95
372.2   285.6   78.1   82.3   13.60   18.68   98.01   12.56   87.5
order   LA5     CP8
```

The message contains 10 integer values, 9 real values, and 3 string values. Because there are more real and integer values than default fields in the send message buffer (the default is eight of each type), you must increase the number of fields for those types of data. You can use the `MMSyncSendMessageNumDatas` function to change the number of fields as shown below:

```
begin model ready function
  call MMSyncSendMessageNumDatas(10,9,3,6)
  /* The first parameter defines the number of integers;
     the second defines the number of reals;
     the third defines the number of strings;
     and the fourth defines the length of each string
     (the number of characters each string field can
     contain). */
end
```

The function changes the number of fields in the send message buffer to 10 integer fields, 9 real fields, and 3 string fields of 6 characters each (more than long enough for each string value in the message). The function is called at the beginning of the simulation. Assuming that no message contains more than 10 integer values, 9 real values, and 3 string values of 6 characters each, you can use this message structure for all messages sent throughout the simulation.

## Defining the content of an MMS message

By default, fields in the send message buffer do not contain any data. After defining the correct number of fields for the message you want to send, you must define values for the fields in the message. The field values compose the content of the message.

You define the content of an outgoing MMS message by calling AutoMod functions to write values to the fields in the send message buffer. The function you use to write a value to a field depends on the field type:

**To define a value for a field of this type:    Use this function:**

|         |   |
|---------|---|
| Integer | MMSyncSendMessageInteger(<Index>,<Value>) |
| Real    | MMSyncSendMessageReal(<Index>,<Value>)    |
| String  | MMSyncSendMessageString(<Index>,<Value>)  |

Each of these functions accepts two parameters. The first parameter is an index (an integer) that identifies the number of the field to which you want to write data. The second parameter is the field value. For more information, see the description of each function in appendix D, “Multi-Model Synchronization Functions.”

**Defining the type of an MMS message**

You can categorize MMS messages by defining a numeric type for each message. An *MMS message type* is an integer greater than or equal to 5000 that is used to identify the purpose and structure of incoming messages.

MMS message types make message interpretation easier. For example, assume that during a simulation, a model receives messages that contain two real values. In some messages, the real values provide new vehicle velocity and acceleration rates. However in other messages, the values provide new conveyor section velocity and acceleration rates. In the receiving model, you have no way of knowing, from the data alone, whether the values should be applied to vehicles or conveyor sections. By assigning a numeric type to distinguish the two messages, you know how to interpret the data. For example, type 5100 messages represent vehicle rates, while type 5200 messages represent section rates.

Message types also make it easy to anticipate the number of values contained in an incoming message. For example type 6000 messages may contain two real values, one integer value and one string value, while type 6100 messages contain three real values, two integer values and no string values. The receiving model only needs to read the type of the message to determine how many values to read from the message. For more information, see “Example – Reading an MMS message” on [page 4.16](#).

To define a type for a message in the send message buffer, use the MMSyncSendMessageType function. If you do not define a message type, the default is 5000.

**Tip** Define a type for each MMS message you send and use a unique number for each new configuration of integer, real, and string values in a message.

If the messages you are sending and receiving during simulation all have the same purpose and number of values (for example, all messages consist of one integer value that is used to increment the same counter) then you do not need to define MMS message types.

## Defining the recipient of an MMS message

Messages that are sent by a model to the MMS Server are broadcast, by default, to each synchronized model in the simulation (including the model from which the message originated). Because the server transmits messages to all models, some models may need to ignore one or more messages from the server (for more information about handling received messages, see “Reading an MMS message” on [page 4.13](#)).

As an alternative to writing code to ignore those messages that do not apply to a model, you can change the recipient of an MMS message to a specific model by calling the `MMSyncSendMessageModel` function before sending the message. The function has one string parameter, which is the name of the model to which the next sent message is broadcast. The MMS Server reads the path to the specified model from the model data file (see “Defining the model data file” on [page 4.19](#)).

---

**Note** You can only specify one recipient model for the message. After the message is sent, the recipient for future messages is reset to all models in the simulation.

---

For example, the following function call indicates that the next message sent by the model will be copied by the server and transmitted only to the model MShop.

```
call MMSyncSendMessageModel("MShop")
```

For more information about the `MMSyncSendMessageModel` function, see “`MMSyncSendMessageModel` function” on [page D.10](#).

## Printing the content of the send message buffer

You can print the current content of the send message buffer to a string using the `MMSyncSendMessagePrint` function. The function returns a tab-delimited string containing the following information:

- The current simulation time
- The message type
- The values of integer fields in the send message buffer
- The values of real fields in the send message buffer
- The values of string fields in the send message buffer

If the trace level of a model or the MMS Server is equal to or greater than one, the `MMSyncSendMessagePrint` file is automatically called after the `MMSyncSendMessage` function, and the return value is written to a trace file. For more information, see “`MMSyncSendMessagePrint` function” on [page D.15](#).

## Sending an MMS message

After defining the structure, content, type, and recipient of the message (optional), you can use the `MMSyncSendMessage` function to send the message to the MMS Server. After the message is sent, the `MMSyncSendMessageClear` function is automatically called to clear the message type and field values in the buffer, and to reset the message recipient to all models in the simulation; however, the structure (the number of fields for each type of data) remains unchanged for future MMS messages. For more information, see “MMSyncSendMessage function” on [page D.8](#).

### Example – Sending an MMS message

This example demonstrates how to send MMS messages that you can use to create or order loads in synchronized models. This example only describes how to send messages from the originating model. For a description of how to read and interpret the messages in the receiving models, see “Example – Reading an MMS message” on [page 4.16](#) and “Example – Interpreting an MMS message” on [page 4.17](#).

This model sends two types of messages: 6000 and 6100. The function `F_CreateLoads` is used to send messages of type 6000, which create loads in the receiving model. The message consists of three types of data: two strings and one integer. The first string, `create`, is a comment describing the purpose of the message. The string is not read by the receiving models, but makes the trace file that records MMS messages easier to read (see “Example – Trace file format” on [page 4.27](#)). The second string defines the type of load to create and the integer value defines the number of loads to create. Because the `MMSyncSendMessageModel` function is not called before the message is sent, this message is broadcast to all models in the simulation.

```
begin F_CreateLoads
    set V_ltypeStr to ArgLtype
    /* Convert the load type that is passed to the function
       to a string value. V_ltypeStr is a variable of type
       string. ArgLtype is a parameter of type LoadType. */

    call MMSyncSendMessageType(6000)
    /* Defines the message type as 6000. */

    call MMSyncSendMessageString(1, "create")
    /* The first parameter identifies the field;
       the second parameter defines the field's value. */
    call MMSyncSendMessageString(2, V_ltypeStr)
    /* The string is written to the second string field. */
    call MMSyncSendMessageInteger(1, ArgQuant)
    /* The message content is now defined. */

    set V_size to MMSyncSendMessage()
    if V_size = 0 then begin
        print "Warning! Message send failed" to message
    return 0
```



```

        end
    return 1
end

```

The integer variable `v_size` is set to the return value of the `MMSyncSendMessage` function. If the variable is equal to zero (indicating that the send could not be completed), then a warning is displayed in the Messages dialog box.

The function `F_OrderLoads` sends type 6100 messages that are used to order loads from an order list in the receiving model. In this case, type 6100 messages consist of only two types of data: one string and one integer. The string value, `order`, is a comment describing the purpose of the message. The integer value defines the number of loads to order from the order list. The `MMSyncSendMessageModel` function is called to set the MMS Server to broadcast the message only to model `model2`.

```

begin F_OrderLoads
    call MMSyncSendMessageType(6100)
        /* Defines the message type as 6100. */

    call MMSyncSendMessageString(1, "order")
        /* The first parameter identifies the field;
           the second parameter defines the field's value. */
    call MMSyncSendMessageInteger(1, ArgQuant)
        /* The message content is now defined. */

    call MMSyncSendMessageModel("model2")
        /* The message is broadcast by the MMS Server only to the
           model named model2. */

    set v_size to MMSyncSendMessage()
    if v_size = 0 then begin
        print "Warning! Message send failed" to message
        return 0
    end
    return 1
end

```

The integer variable `v_size` is set to the return value of the `MMSyncSendMessage` function. If the variable is equal to zero (indicating that the send could not be completed), then a warning is displayed in the Messages dialog box.

## Reading an MMS message

Each time the MMS Server broadcasts a message to a model, the message is automatically stored in the read message buffer and the `model mmsyncmessage` function is called. The *read message buffer* is a temporary storage area in RAM where the structure, type, and content of an incoming MMS message is stored. The `model mmsyncmessage` function is a function that is called automatically each time a message is received.

You can call functions in the `model mmsyncmessage` function to read the following:

- The message structure.
- The message type.
- The content of the message.
- The time at which the message was sent from the originating model.

Using functions to read information from the read message buffer is described in detail in the following sections.

**! Important** The functions that are used to read from the read message buffer can only be called within the `model mmsyncmessage` function.

### Reading the structure of an MMS message

You can read the structure of the message contained in the read message buffer by calling AutoMod functions to determine the number of integer, real, and string fields in the message.

**Tip** You do not need to read the structure of incoming messages if either of the following is true:

- All received MMS messages have the same structure. When models are communicating with only one message structure, then the number of values to read for each type of data doesn't change between messages.
- Incoming MMS messages have been categorized using message types. If the sending model defines a unique message type for each new configuration of real, integer, and string values in a message, then you can use the message type to determine how many values to read from the message. For information on defining message types, see "Defining the type of an MMS message" on [page 4.10](#).

The function you use to read the number of fields in the message depends on the field type:

| To read the number of fields for this type of data: | Use this function:                          |
|---|---|
| Integer   | <code>MMSyncReadMessageNumIntegers()</code> |
| Real  | <code>MMSyncReadMessageNumReals()</code>    |
| String  | <code>MMSyncReadMessageNumStrings()</code>  |

You can also read the length of the string fields in the message using the `MMSyncReadMessageStringLength` function. For more information about each of these functions, see appendix D, "Multi-Model Synchronization Functions."

## Reading the content of an MMS message

You can read the content of an incoming MMS message by calling AutoMod functions to read values from the fields in the read message buffer. The function you use to read a value depends on the field type:

**To read a value from a field of this type:**      **Use this function:**

|         |  |
|---------|--|
| Integer | <code>MMSyncReadMessageInteger(&lt;Index&gt;)</code> |
| Real    | <code>MMSyncReadMessageReal(&lt;Index&gt;)</code>    |
| String  | <code>MMSyncReadMessageString(&lt;Index&gt;)</code>  |

Each of these functions accepts an integer parameter that identifies the index number of the field from which you want to read data, and returns the field's value. For more information about each of these functions, see appendix D, "Multi-Model Synchronization Functions."

## Reading the type of an MMS message

To read the type defined for the message in the read message buffer, use the `MMSyncReadMessageType` function. MMS message types allow you to determine the structure of an incoming message, without needing to call functions to read the number of fields in each message. For an example of using message types to read values from a message, see "Example – Reading an MMS message" on [page 4.16](#).

## Reading the time that an MMS message was sent

You can read the time at which a received message was sent from the originating model using the `MMSyncReadMessageTime` function (see "MMSyncReadMessageTime function" on [page D.7](#)).

Reading the time a message was sent is useful if models are periodically synchronized and you need to schedule a function to interpret the message at a specific time after the message was sent. If models are exactly synchronized, you do not need to use the `MMSyncReadMessageTime` function, because messages are received at the same time that they are sent by the originating model. For more information about exact and periodic synchronization, see "Synchronizing simulation clocks" on [page 4.4](#).

## Printing the content of the read message buffer

You can print the current content of the read message buffer to a string using the `MMSyncReadMessagePrint` function. The function returns a tab-delimited string containing the following information:

- The time the current message was received
- The message type
- The values of integer fields in the read message buffer

- The values of real fields in the read message buffer
- The values of string fields in the read message buffer

If the trace level of a model or the MMS Server is equal to two, the `MMSyncReadMessagePrint` file is automatically called before the `model mmsyncmessage` function, and the return value is written to a trace file. For more information, see “[MMSyncReadMessagePrint function](#)” on [page D.4](#).

**Example – Reading an MMS message** In this example, the `model mmsyncmessage` function is called to read messages from the read message buffer. The `model mmsyncmessage` function is called automatically each time a message is received from the MMS Server.

The function first reads the type of message contained in the read message buffer and then reads values from the message fields. Because message types are used, it is not necessary to read the number of fields contained in the message before reading the message content.

```
begin model mmsyncmessage function
  if MMSyncReadMessageType() is 6000 then begin
    /* type-6000 messages are used to create loads */
    set V_Quant to MMSyncReadMessageInteger(1)
    set V_LtypeStr to MMSyncReadMessageString(2)
    call F_create(V_Quant, V_LtypeStr)
  end

  else if MMSyncReadMessageType() is 6100 then begin
    /* type-6100 messages are used to order loads from an
       order list */
    set V_Quant to MMSyncReadMessageInteger(1)
    call F_order(V_quant)
  end

  else if MMSyncReadMessageType() is 7000 then begin
    /* type-7000 messages are not used by this model */
  end

  else begin
    print "Received unknown message type "
      MMSyncReadMessageType() " - ignoring message"
      to message
  end

  return true /* return value not used */
end
```

The values that are read from the message are passed to a function that performs actions based on the message content (for more information, see “[Example – Interpreting an MMS message](#)” on [page 4.17](#)). The function that is called to interpret the message varies

depending on the message type (for example, type 6000 messages call the `F_create` function, type 6100 messages call the `F_order` function, and so on).

---

**Tip** Because MMS messages are broadcast, by default, to each synchronized model in the simulation (including the model from which the message originated) you may need to ignore some messages that are received from the MMS Server. For example, the receiving model in this example could send messages of type 7000, which are not interpreted when the messages are sent back by the server. As an alternative, you can call the `MMSyncSendMessageModel` function before sending a message to cause the message to be sent to a specific model (see “Defining the recipient of an MMS message” on [page 4.11](#)).

---

## Interpreting an MMS message

Interpreting a message is the process of writing user-defined functions or procedures that define which actions should be performed based on the message content. After reading values from an incoming MMS message in the `model mmsyncmessage` function, you can call custom functions to use the values during the simulation. For example, you can dynamically change the value of vehicle or section velocities using new values received in a message, or you could bring down or bring up resources when messages are received from another model. The following example shows one method of interpreting values received in an MMS message.

### Example – Interpreting an MMS message

In this example, a model receives messages from the MMS Server that vary by type. When a message is received, a function is called to perform actions using the values that are read from the message. This example shows the functions that are used to interpret values read by the `model mmsyncreadmessage` function defined in a previous example (see “Example – Reading an MMS message” on [page 4.16](#)).

```
begin F_create function
  /* called for messages of type 6000 */
  set V_ltype to ArgLtypeStr
  /* converts the string expression to a load type */
  create ArgQuant loads of load type V_ltype to P_start
  return true /* return value not used */
end

begin F_order function
  /* called for messages of type 6100 */
  order ArgQuant loads from OL_supply to continue
  in case order not filled backorder on OL_back
  return true /* return value not used */
end
```

The `F_create` function creates new loads in the receiving model. The load type variable `v_ltype` is set to the parameter `ArgLtypeStr`, which converts the message's string expression to an AutoMod load type. The integer expression that is read from the message defines how many loads are created.

The `F_order` function is used to order loads on an order list to continue. The integer expression that is read from the message defines how many loads are ordered off the list.

---

## Starting a synchronized simulation

After writing code to transfer MMS messages between synchronized models, you are ready to start the synchronized simulation.

### ► To start the synchronized simulation

1. Set up remote machines to conduct model runs (see “Setting up remote machines” on [page 4.18](#)).
2. Define the model data file (see “Defining the model data file” on [page 4.19](#)).
3. Start the MMS Server (see “Starting the MMS Server” on [page 4.22](#)).

The MMS Server automatically runs the models defined in the model data file and begins the simulation.

---

**Note** You can reproduce an individual model's activity during a synchronized simulation by running one model with a trace file from the simulation. For more information, see “Running a model with a trace file” on [page 4.28](#).

---

## Setting up remote machines

If you want to synchronize models that are running on a different machine than the MMS Server, each remote machine must have the Process Server software and a full or runtime version of AutoMod (authorized for model communications) installed.

### ► To set up a machine to conduct remote runs

1. Build a model on the remote machine. If the remote machine only has a runtime version of AutoMod installed, you must copy a built model to the remote machine (built models consist of a `.dir` directory, a model executable, and any external data files). Remember the path of the built model on the remote machine because you will need it when defining the model data file.

2. Start the Process Server on the remote machine.

---

**Note** For information on starting the Process Server, select the Process Server help file from the **Start** menu.

---

The machine is now set up to make remote runs.

## Defining the model data file

The *model data file* is a tab-delimited text file that defines the location and command line options for model executables and the name and level of each model's trace file. The MMS Server uses the model data file to determine which models to run during a simulation. An example model data file is shown below:

| NAME   | EXECUTABLE           | ARGUMENTS | MACHINE         | PORT | TRACELEVEL | TRACEFILE |
|--------|----------------------|-----------|-----------------|------|------------|-----------|
| Demo1  | model\demo1          | -iL       | localhost       |      | 2          | trace.txt |
| Demo2  | c:\users\class\demo2 | -i        | class1.acme.com | 7070 | 2          | trace.txt |
| Demo3  | c:\users\class\demo3 | -i        | class2.acme.com | 7070 | 2          | trace.txt |
| ~Demo4 | c:\users\class\demo4 | -i        | class3.acme.com | 7070 | 2          | trace.txt |

### ► To define the model data file

1. Open a text editor or spreadsheet program.
2. On the first line of a new file, type the following headers (separated by tabs).

```
NAME EXECUTABLE ARGUMENTS MACHINE PORT TRACEFILE TRACELEVEL
```

**!** **Important** Use only one tab between each header.

3. Add a line to the file for each model in the simulation. Each line must include information about the model in the format described in the table that follows this procedure.

---

**Tip** Adding a tilde (~) to the beginning of a line causes the MMS Server to ignore that line of the model data file. Using a tilde is a good way to add comments to the file or temporarily remove models from the simulation.

---

4. Save the file as a tab-delimited text file.

You can save the file in any directory and give it any name you want. By default, the MMS Server looks for a trace file named `models.mms` in the directory from which you start the MMS Server. If you select a different name or location for the model data

file, you will need to define the file name and location when starting the MMS Server (for more information, see “Starting the MMS Server” on [page 4.22](#)).

| Field      | Description  |
|------------|--|
| NAME       | <p>A custom name for the model (the name does <i>not</i> need to match the name of the model in AutoMod). Defining a custom name for a model can be useful if you have two models with the same name running on different machines.</p> <p><i>Default:</i> <code>MMSmodel#</code> where # is an integer that starts at one and increments for each model in the model data file.</p>   |
| EXECUTABLE | <p>The path to the model executable. This can be a full path (for example, <code>c:\demos\models\model1\model1.exe</code>) or a relative path beginning with the directory from which the Process Server is started (for remote runs) or from which the MMS Server is started (for local runs); for example <code>model1\model1.exe</code> assuming that the server was started from the models directory.</p> <p><b>Tip</b> You do not need to include the .exe extension in the name of the model executable.</p> <p><i>Default:</i> None. The location of the model executable must be defined for each model in the data file.</p>   |
| ARGUMENTS  | <p>The command line options that are used to start the model (for example, use <code>-w</code> to run without displaying any dialog boxes and <code>-i</code> to run without requiring any interaction from the user). For a complete list of the AutoMod command line options, see the “Running a Model” chapter in the <i>AutoMod User’s Guide</i>.</p> <p><i>Default:</i> None. Models that are started by the MMS Server are paused and must be manually continued to begin the simulation.</p> <p><b>Tip</b> To avoid the need to manually continue each model running on a remote machine, use the <code>-i</code>, <code>-n</code>, or <code>-w</code> command line options to start the simulation without user interaction.</p> |
| MACHINE    | <p>The network name of the machine that is conducting the model run during simulation. If the MMS Server and the model are running on the same machine, you can use the keyword <code>localhost</code> for the machine name. If you want to run the model on a different machine than the MMS Server, you must build the model and start the Process Server on the remote machine (for more information, see “Setting up remote machines” on <a href="#">page 4.18</a>).</p> <p><i>Default:</i> <code>localhost</code>.</p>  |



| Field      | Description   |
|------------|---|
| PORT       | <p>The port number on which the Process Server is communicating (for remote model runs only). This number is defined when you start the Process Server. If the model is running on the same machine as the MMS Server, you can omit the port number.</p> <p><i>Default:</i> 7070.</p>   |
| TRACEFILE  | <p>The name of a text file where the model's MMS messages are printed. You can create a separate trace file for each model that is participating in the simulation. You can also create a trace file for messages that are received and/or sent by the MMS Server (for more information, see "Starting the MMS Server" on <a href="#">page 4.22</a>). Trace files are useful for debugging and can also be used to reproduce a model's activity during a synchronized simulation (for more information, see "Running a model with a trace file" on <a href="#">page 4.28</a>). The trace file for each model is saved in the model's directory.</p> <p><i>Default:</i> MMSmodel.trace.</p>  |
| TRACELEVEL | <p>The trace level for messages sent and received by the model during the simulation. The <i>trace level</i> is an integer that determines which messages are written to the trace file:</p> <ul style="list-style-type: none"> <li>• <b>0</b> - A trace file is not created.</li> <li>• <b>1</b> - MMS messages that are sent from this model are printed to the trace file.</li> <li>• <b>2</b> - MMS messages that are sent from or received by this model are printed to the trace file.</li> </ul> <p><i>Default:</i> 0 (messages are not traced).</p> <p><b>Note</b> If you want to use the trace file to reproduce simulation results, the trace level must be greater than zero (for more information, see "Running a model with a trace file" on <a href="#">page 4.28</a>).</p> |

## Example - Defining the model data file

This example shows a model data file that is used by the MMS Server to start three models in a simulation:

| NAME   | EXECUTABLE           | ARGUMENTS | MACHINE         | PORT | TRACELEVEL | TRACEFILE |
|--------|----------------------|-----------|-----------------|------|------------|-----------|
| Demo1  | model\demo1          | -iL       | localhost       |      | 2          | trace.txt |
| Demo2  | c:\users\class\demo2 | -i        | class1.acme.com | 7070 | 2          | trace.txt |
| Demo3  | c:\users\class\demo3 | -i        | class2.acme.com | 7070 | 2          | trace.txt |
| ~Demo4 | c:\users\class\demo4 | -i        | class3.acme.com | 7070 | 2          | trace.txt |

The first model, demo1, is running on the same machine as the MMS Server. Each of the other models in the simulation are running on remote machines. Notice the tilde at the beginning of the line defining the location of model Demo4. The tilde causes the line to be ignored by the server, omitting Demo4 from the simulation.

All of the models in the simulation are passed the **-i** command line option to allow the simulation to start without user interaction at each machine. The **-L** command line option for demo1 causes the graphics in that model to display in wireframe mode.

All of the models are writing a trace file that contains all of the MMS messages that are sent and received by the models. The trace file can be used later for debugging or to reproduce the results of the simulation while running only one model (for more information, see “Running a model with a trace file” on [page 4.28](#)). To view a list of the messages that are received and sent by a model during the synchronized simulation, you can open its trace file with any text editor (for more information, see “Trace file format” on [page 4.25](#)).

## Starting the MMS Server

When you start the MMS server, it automatically locates and runs each model that is participating in the simulation.

### ► To start the MMS Server

1. Open a command prompt and navigate to the directory from which you want to start the MMS Server.

---

**Note** The directory from which you start the MMS Server determines:

- The location where the MMS Server’s trace file is saved.
- The default directory where the server attempts to locate the model data file.
- The base directory for locating model executables that are defined in the model data file using a relative path (for more information, see “Defining the model data file” on [page 4.19](#)).

2. Type the following to start the MMS Server:

```
MultiModelSync [command line options] [model_data_file_path]
```

## Command line options

If you start the MMS Server without typing any command line options, the server uses the default values for each option. The command line options are defined in the following table.

---

**Note** Command line options are case-sensitive.

---

| Option             | Description  |
|--------------------|--|
| -p [portnumber]    | A number greater than 1000 that defines which port the models use to connect to the MMS Server. Use either the <b>-p</b> or <b>-s</b> option (do not use both).<br><br><i>Default:</i> -p 5502   |
| -s [servicename]   | The service name for the port that models use to connect to the MMS Server. You must define a service name in the services file before you can use this option. For more information, see “Defining a service name” on <a href="#">page 3.6</a> . Use either the <b>-p</b> or <b>-s</b> option (do not use both).  |
| -me                | Sets the MMS Server to exact synchronization (see “Exact synchronization” on <a href="#">page 4.4</a> ). Use either the <b>-me</b> or <b>-mp</b> option (do not use both).<br><br><i>Default:</i> -me  |
| -mp [time]         | Sets the MMS Server to periodic synchronization (see “Periodic synchronization” on <a href="#">page 4.6</a> ). The time you define is the length of time (in seconds) that models run asynchronously between synchronization updates. Use either the <b>-me</b> or <b>-mp</b> option (do not use both).  |
| -f [tracefilename] | The name of a text file where the server’s MMS messages are printed. You can also create a separate trace file for each model that is participating in the simulation (for more information, see “Defining the model data file” on <a href="#">page 4.19</a> ). Trace files are useful for debugging and can also be used to reproduce a model’s activity during a synchronized simulation (for more information, see “Running a model with a trace file” on <a href="#">page 4.28</a> ). The server’s trace file is saved in the directory in which you started the MMS Server.<br><br><i>Default:</i> -f MMSserver.trace |

---

| Option          | Description   |
|-----------------|---|
| -t [tracelevel] | <p>The trace level for messages sent and received by the MMS Server during the simulation. The trace level is an integer that determines which messages are written to the trace file:</p> <ul style="list-style-type: none"> <li>• <b>0</b> - A trace file is not created.</li> <li>• <b>1</b> - MMS messages that are sent from this model are printed to the trace file.</li> <li>• <b>2</b> - MMS messages that are sent from or received by this model are printed to the trace file.</li> </ul> <p><i>Default:</i> -t 0 (messages are not traced).</p> <p><b>Note</b> If you want to use the trace file to reproduce a model's activity during the synchronized simulation, the trace level should be greater than zero (for more information, see “Running a model with a trace file” on <a href="#">page 4.28</a>).</p> |

## Model data file path

When starting the MMS Server, the path and file name of the model data file must be passed to the server along with the command line options. The model data file defines the name and location of the models that are participating in the simulation (for more information, see “Defining the model data file” on [page 4.19](#)).

**Default value** If you do not define a name and location for the model data file, the server looks for a model data file named `models.mms` in the directory in which you started the MMS Server. If the model data file is not found, the server cannot start.

**Example - Starting the MMS Server** In this example, the MMS Server is started using the model data file “`model.dat`.”

```
MultiModelSync -p 4000 -mp 5 -t 1 -f trace.mms
c:\models\model.dat
```

The server is started using port number 4000. The server periodically synchronizes models every five seconds and copies sent messages to the file `trace.mms`.

To view a list of the messages that are received and sent by the MMS Server during the synchronized simulation, you can open the trace file with any text editor (for more information, see “Trace file format” on [page 4.25](#)).

---

## Using trace files

A *trace file* is a tab-delimited text file that contains printed copies of MMS messages that were sent or received during a simulation. Trace files can be created by the MMS Server and/or by individual models that participated in the synchronized simulation. The file's trace level (0, 1 or 2) determines which messages are printed to the file. For information about creating a trace file using the MMS Server, see “Command line options” on [page 4.23](#). For information about creating a trace file using an individual model in the simulation, see “Defining the model data file” on [page 4.19](#).

Trace files are useful for model debugging. A trace file allows you to see the time at which messages were sent or received and the values contained in the messages. Trace files can also be used to reproduce an individual model's activity, without running each of the models that participated in a synchronized simulation.

### Trace file format

The first line of a trace file displays the type of trace file (either model or server) and the level of the trace.

The second line of the trace file displays the headings for each column of data in the file. The headings are defined in the following table.

| Heading      | Description   |
|--------------|---|
| Type         | <p>The source or destination of the MMS message. The possible values depend on whether the trace file was created by a model or the server.</p> <p>For a model trace file, the possible values are:</p> <ul style="list-style-type: none"><li>• <b>MMSServer</b> - Indicates a message received by the model from the MMS Server.</li><li>• <b>From-Model</b> - Indicates a message sent from the model to the MMS Server.</li></ul> <p>For the server trace file, the possible values are:</p> <ul style="list-style-type: none"><li>• <b>MMSServer</b> - Indicates a message received by the server from a model.</li><li>• <b>To-[model_name]</b> - Indicates a message sent by the server to the specified model.</li></ul> |
| Time         | <p>The time the message was either sent or received. Time values are expressed in seconds.</p>  |
| MMS Type     | <p>The type assigned to the MMS message. For more information, see “Defining the type of an MMS message” on <a href="#">page 4.10</a>.</p>  |
| Message Data | <p>The values contained in the integer, real, and string fields of the message (in that order). Each field value is separated by a tab.</p>   |

## Example – Trace file format

The following example shows a level-two trace file from a model:

| MultiModelSync Model TraceFile |        |          | TraceLevel 2 |           |         |
|--------------------------------|--------|----------|--------------|-----------|---------|
| Type                           | Time   | MMS Type | Message Data |           |         |
| MMSServer                      | 181.00 | 6000     | 7            | “create”  | “small” |
| MMSServer                      | 211.00 | 6000     | 3            | “create”  | “large” |
| MMSServer                      | 241.00 | 6000     | 1            | “create”  | “large” |
| MMSServer                      | 264.57 | 6100     | 8            | “order”   |         |
| MMSServer                      | 271.00 | 6000     | 4            | “create”  | “small” |
| MMSServer                      | 301.00 | 6000     | 1            | “create”  | “large” |
| MMSServer                      | 331.00 | 6000     | 4            | “create”  | “large” |
| MMSServer                      | 361.00 | 6000     | 1            | “create”  | “small” |
| MMSServer                      | 391.00 | 6000     | 3            | “create”  | “small” |
| From-Model                     | 399.00 | 7000     | 3            | 393.12500 | “small” |
| MMSServer                      | 399.00 | 7000     | 3            | 393.12500 | “small” |

The model received three different types of MMS messages from the MMS Server (types 6000, 6100, and 7000). The model sent only one message of type 7000 (at time 399 seconds), which was broadcast back to the model from the MMS Server at the exact same simulation time.

The following example shows the MMS Server's trace file for the same simulation:

| MultiModelSync Server TraceFile |        |          | TraceLevel 2 |           |         |
|---------------------------------|--------|----------|--------------|-----------|---------|
| Type                            | Time   | MMS Type | Message Data |           |         |
| MMSServer                       | 181.00 | 6000     | 7            | "create"  | "small" |
| To-model1                       | 181.00 | 6000     | 7            | "create"  | "small" |
| To-model2                       | 181.00 | 6000     | 7            | "create"  | "small" |
| MMSServer                       | 211.00 | 6000     | 3            | "create"  | "large" |
| To-model1                       | 211.00 | 6000     | 3            | "create"  | "large" |
| To-model2                       | 211.00 | 6000     | 3            | "create"  | "large" |
| MMSServer                       | 241.00 | 6000     | 1            | "create"  | "large" |
| To-model1                       | 241.00 | 6000     | 1            | "create"  | "large" |
| To-model2                       | 241.00 | 6000     | 1            | "create"  | "large" |
| MMSServer                       | 264.57 | 6100     | 8            | "order"   |         |
| To-model2                       | 264.57 | 6100     | 8            | "order"   |         |
| MMSServer                       | 271.00 | 6000     | 4            | "create"  | "small" |
| To-model1                       | 271.00 | 6000     | 4            | "create"  | "small" |
| To-model2                       | 271.00 | 6000     | 4            | "create"  | "small" |
| MMSServer                       | 301.00 | 6000     | 1            | "create"  | "large" |
| To-model1                       | 301.00 | 6000     | 1            | "create"  | "large" |
| To-model2                       | 301.00 | 6000     | 1            | "create"  | "large" |
| MMSServer                       | 331.00 | 6000     | 4            | "create"  | "large" |
| To-model1                       | 331.00 | 6000     | 4            | "create"  | "large" |
| To-model2                       | 331.00 | 6000     | 4            | "create"  | "large" |
| MMSServer                       | 361.00 | 6000     | 1            | "create"  | "small" |
| To-model1                       | 361.00 | 6000     | 1            | "create"  | "small" |
| To-model2                       | 361.00 | 6000     | 1            | "create"  | "small" |
| MMSServer                       | 391.00 | 6000     | 3            | "create"  | "small" |
| To-model1                       | 391.00 | 6000     | 3            | "create"  | "small" |
| To-model2                       | 391.00 | 6000     | 3            | "create"  | "small" |
| MMSServer                       | 399.00 | 7000     | 3            | 393.12500 | "small" |
| To-model1                       | 399.00 | 7000     | 3            | 393.12500 | "small" |
| To-model2                       | 399.00 | 7000     | 3            | 393.12500 | "small" |

Each time the MMS Server receives a message of type 6000 or 7000 from a model, the server broadcasts the message to both models in the simulation. Messages of type 6100 are sent only to the model model2; for information on sending messages to a specific model, see "Defining the recipient of an MMS message" on [page 4.11](#).

## Running a model with a trace file

Running a model with a trace file allows you to reproduce the activity of the model during a synchronized simulation without running each of the simulation's participating models. Running a model with a trace file is useful when you want to present only one part of a synchronized



simulation or when some of the synchronized models are not available. To run a model with a trace file, the level of the trace file must be greater than zero (see “TRACELEVEL” on [page 4.21](#)).

**Important** If you make changes to any model that participated in the simulation, you must create a new trace file for accurate simulation results.

► **To reproduce a model’s activity during a synchronized simulation**

1. Define the model data file to create a trace file for the model you want to run independently (see “Defining the model data file” on [page 4.19](#))

---

**Tip** You can use either the trace file generated by a model or the MMS Server to reproduce a model’s activity during a synchronized simulation. For information on defining a trace file for the MMS Server, see “Command line options” on [page 4.23](#).

---

2. Run the simulation with the MMS Server to create the trace file (see “Starting the MMS Server” on [page 4.22](#)).
3. From a command prompt, navigate to the directory of the model executable and type the following:

```
<model_executable_name> -MMS file=<trace_file_name>
```

For example, to start the model demo1 using the trace file trace.txt, type:

```
demo1 -MMS file=trace.txt
```

The simulation begins and the model runs exactly as it did during the synchronized simulation.

## Generating a new trace file with a different trace level

While running a model with a trace file, you can generate a new trace file with a different trace level. The generated trace file exactly matches the trace file that is used to start the simulation, except that it contains more or fewer messages depending on the new trace level.

► **To generate a trace file with a different trace level**

From a command prompt, navigate to the directory of the model executable and type the following:

```
<model_executable_name> -MMS file=<trace_file_name>,  
tracefile=<new_trace_file_name>,trace=<new_trace_level>
```

The commands must be separated from the original trace file name with a comma, and must not contain any spaces, for example:

```
demo1 -MMS file=trace.txt,tracefile=newtrace.txt,trace=1
```

The simulation is started using the existing trace file, trace.txt. The simulation generates a new, level-one trace file named newtrace.txt. The new trace file is saved in the same directory as the original trace file.



# OPC Quality Codes

This appendix describes the OPC quality codes that OPC servers provide for each item value that is read by a client application. The codes provide information about the reliability of the item value.

---

## Codes -1 through 28 (“Bad” or unreliable data)

Quality codes -1 through 28 indicate “bad” or unreliable data:

| Code | Description. |
|------|--------------|
|------|--------------|

- |    |  |
|----|--|
| -1 | The item whose value you are trying to read cannot be found on the specified server.   |
| 0  | The data is unreliable for an unspecified reason.  |
| 4  | The data is unreliable because there is a server-specific problem with the configuration. For example, the item whose value you are reading has been deleted from the configuration.   |
| 8  | The data is unreliable because the item should be logically connected to an input, but is not. This quality value might indicate that no value is available at this time, because the value has not been provided by the data source.  |
| 12 | The data is unreliable because a device failure has been detected.   |
| 16 | The data is unreliable because a sensor failure has been detected.   |
| 17 | The data is unreliable because a sensor failure has been detected; the value is fixed at a low limit.  |
| 18 | The data is unreliable because a sensor failure has been detected; the value is fixed at a high limit.   |
| 19 | The data is unreliable because a sensor failure has been detected; the value is fixed at a constant amount.  |
| 20 | The data is unreliable because communication has failed; however, the last known value is available. You can determine the time at which the value was last updated by referring to the item’s timestamp (the item’s timestamp is copied into the <code>OPCTimestamp</code> variable each time the model reads an item’s value). |
| 24 | The data is unreliable because communication has failed; there is no last known value for the item.  |
-

---

**Code Description.**

- |    |  |
|----|--|
| 28 | The data is unreliable because it is no longer being scanned or is otherwise locked. This quality value might indicate that the active state of the item or the group containing the item is inactive or out of service. |
|----|--|

---

## Codes 64 through 91 (“Uncertain” data)

Quality codes 64 through 91 indicate “uncertain” data:

---

**Code Description**

- |    |  |
|----|--|
| 64 | The data’s reliability is uncertain for an unspecified reason.   |
| 68 | The data’s reliability is uncertain because whatever data source was writing this item value hasn’t updated the value within an acceptable period of time; consequently, the item’s value might be outdated. You can determine the time at which the value was last updated by referring to the item’s timestamp (the item’s timestamp is copied into the <code>OPCTimestamp</code> variable each time the model reads an item’s value). |
| 69 | The data’s reliability is uncertain because whatever data source was writing this item value hasn’t updated the value within an acceptable period of time; the value is fixed at a low limit.  |
| 70 | The data’s reliability is uncertain because whatever data source was writing this item value hasn’t updated the value within an acceptable period of time; the value is fixed at a high limit.   |
| 71 | The data’s reliability is uncertain because whatever data source was writing this item value hasn’t updated the value within an acceptable period of time; the value is fixed at a constant amount.  |
| 80 | The data’s reliability is uncertain because the sensor is known to be out of calibration through some form of internal diagnostics.  |
| 81 | The data’s reliability is uncertain because the value has become fixed at the sensor’s low limit.  |
| 82 | The data’s reliability is uncertain because the value has become fixed at the sensor’s high limit.   |
| 85 | The data’s reliability is uncertain because the value is outside the low limit defined for the item; this does not imply that the value cannot move further out of range.  |
| 86 | The data’s reliability is uncertain because the value is outside the high limit defined for the item; this does not imply that the value cannot move further out of range.   |
-

| Code | Description |
|------|-------------|
|------|-------------|

|    |  |
|----|--|
| 88 | The data's reliability is uncertain because the value is derived from multiple data sources and has less than the required number of "good" sources. |
|----|--|

|    |   |
|----|---|
| 89 | The data's reliability is uncertain because the value is derived from multiple data sources and has less than the required number of "good" sources; the value is fixed at a low limit. |
|----|---|

|    |  |
|----|--|
| 90 | The data's reliability is uncertain because the value is derived from multiple data sources and has less than the required number of "good" sources; the value is fixed at a high limit. |
|----|--|

|    |  |
|----|--|
| 91 | The data's reliability is uncertain because the value is derived from multiple data sources and has less than the required number of "good" sources; the value is fixed at a constant limit. |
|----|--|

---

## Codes 192 through 219 ("Good" or reliable data)

Quality codes 192 and above indicate "good" or reliable data:

| Code | Description |
|------|-------------|
|------|-------------|

|     |                       |
|-----|-----------------------|
| 192 | The data is reliable. |
|-----|-----------------------|

|     |  |
|-----|--|
| 195 | The data is reliable; the value is fixed at a constant amount. |
|-----|--|

|     |  |
|-----|--|
| 216 | The data is reliable; the value has been overridden. Typically this quality code indicates that the item's input has been disconnected and that the item's value has been set to a manually entered value. |
|-----|--|

|     |  |
|-----|--|
| 219 | The data is reliable; the value has been overridden and is fixed at a constant amount. Typically this quality code indicates that the item's input has been disconnected and that the item's value has been set to a manually entered value. |
|-----|--|





# OPC Communication Functions

This appendix describes the functions that you can use when building an emulation model that communicates with one or more OPC servers. The appendix also provides syntax diagrams for referring to a machine or item in a function parameter (see “Defining a reference to a machine” on [page B.26](#) and “Defining a reference to an item” on [page B.26](#)).

---

**Note** To use the functions listed in this appendix, you must purchase and authorize either the Model Communications module or the Model Communications Plus module.

---

---

## System functions

System functions are predefined functions that you can call in a model to manage connections and communicate with an OPC server during a model run. Each function’s explanation includes the function’s name and required parameters, a description of how to use the function, and the function’s return value.

---

**Note** Because connections to OPC servers are not opened until the model is running, AutoMod does not provide syntax checking to verify whether a machine, server, group, or item name is valid. Consequently, it is important to write code within the model to check the return value of OPC functions and verify that they are successful.

---

## MilliSleep function

### Description

The `MilliSleep` function allows you to pause a model for the specified number of milliseconds (measured in real time, as opposed to simulation time). The actual time paused is accurate to within 10 milliseconds. For more information, see “Creating delays measured in real time” on [page 2.44](#).

### Parameters

The `MilliSleep` function requires one parameter:

`MilliSleep(<Time>)`

The parameter is defined in the following table:

| Parameter | Type    | Description   |
|-----------|---------|---|
| Time      | Integer | The number of milliseconds, measured in real time, that the model pauses. |

### Return value

You can ignore the integer return value; this function always returns zero.

## OPCAddGroup function

### Description

The `OPCAddGroup` function defines a group on the specified server. You can use groups to organize items that have the same update rate and deadband value. A group's update rate and deadband value allow you to conserve resources on the server by limiting how frequently the server scans items for changed values and sends updated item values to the model. For more information, see "Group properties" on [page 2.6](#) and "Adding groups to a server" on [page 2.16](#).

When creating groups on a server, you can automatically subscribe to all items in the group. For more information, see "Subscribing to items when adding groups to a server" on [page 2.17](#).

### Parameters

The `OPCAddGroup` function requires four parameters:

`OPCAddGroup(<Server>, <Group>, <Rate>, <Deadband>)`



The parameters are defined in the following table:

| Parameter | Type   | Description   |
|-----------|--------|---|
| Server    | String | The name or alias of the OPC server on which you want to add the group ( for information on how to define a server's alias, see "OPCConnectServer function" on <a href="#">page B.5</a> ).  |
| Group     | String | The name of the group you are adding to the server.<br><b>Note</b> You can automatically subscribe to all items in the group by appending the keyword [subscribe] (enclosed in brackets) to the group name.                             |
| Rate      | Real   | The update rate in milliseconds; this rate determines how frequently the server scans subscribed item values in the group to determine if the values have changed.  |
| Deadband  | Real   | The deadband expressed as a percentage from 0.0 to 100.0; the deadband defines the percentage that a subscribed item's value must change, based on its possible range of values, before the server sends an updated value to the model. |

## Return value

An integer value that indicates the success of the function:

- 0 The group was successfully added to the server.
- 4 The connection to the server does not exist or the server name was `null`. Verify that the connection to the OPC server is open (see "Verifying connections" on [page 2.22](#)) and try using the server's alias to identify the server.
- 5 The server refused to add the group.

## OPCAddItem function

### Description

The `OPCAddItem` function adds an item to a group on the server. After adding an item to a group, the model can read and write the item's value during a model run.

When adding an item to a group, you can also link the item with a variable in the AutoMod model. Linked variables reduce the number of function calls required in the model by providing automatic updates to

the model or OPC server when item values change. The types of links you can create are: subscribe, update, and set. For more information, see “Adding items to a group” on [page 2.17](#).

## Parameters

The `OPCAddItem` function requires four parameters:

`OPCAddItem(<Server>, <Group>, <Item>, <ItemType>)`

The parameters are defined in the following table:

| Parameter | Type   | Description   |
|-----------|--------|---|
| Server    | String | The name or alias of the OPC server that contains the item you want to add to a group (for information on how to define a server’s alias, see “OPCConnectServer function” on <a href="#">page B.5</a> ).  |
| Group     | String | The name of the group to which you want to add the item (for information on how to add a group to a server, see “Adding groups to a server” on <a href="#">page 2.16</a> ). If the group does not already exist on the server, the function creates the group with an update rate of 1000 milliseconds and a deadband of zero (for more information, see “Group properties” on <a href="#">page 2.6</a> ).                                      |
| Item      | String | <p>The name of the item you are adding to a group. If the item does not already exist on the server, the function attempts to create the item and add it to the assigned group (for more information, see “Creating items on an OPC server” on <a href="#">page 2.18</a>).</p> <p><b>Note</b> You can link OPC items with variables in an AutoMod model by appending a link type and variable name, enclosed in brackets, to the item name.</p> |
| ItemType  | String | <p>The item’s type. Valid types are:</p> <ul style="list-style-type: none"><li>• <b>Integer</b> - Use this type for integer or Boolean items.</li><li>• <b>Real</b> - Use this type for float items.</li><li>• <b>String</b> - Use this type for string or character items.</li></ul> <p><b>Note</b> These strings are <i>not</i> case sensitive.</p>   |

## Return value

An integer value that indicates the success of the function:

- 0 The item was successfully added to the group.
- 4 The connection to the server does not exist or the server name was `null`. Verify that the connection to the OPC server is open (see “Verifying connections” on [page 2.22](#)) and try using the server’s alias to identify the server.
- 5 The server refused to add the item. Check the function’s parameters to make sure that the item name and group name are both correct.


## OPCConnectServer function

### Description

The `OPCConnectServer` function opens a connection to a local or remote OPC server.

**Important** When opening connections to servers at your facility, ensure that you connect only to servers that are dedicated to testing. Connecting and writing values to a production server could result in system hardware damage or operator injury.

After opening a connection, you can create groups on the server and add items to each group (for more information, see “Opening a connection using functions” on [page 2.12](#)). After you have opened a connection, it remains open until you close it in one of the following ways:

- By calling the `OPCDisconnect` function (for more information, see “`OPCDisconnect` function” on [page B.6](#)).
- By closing the AutoMod runtime environment (either by clicking **File > Exit** or by clicking **Close**  in the Simulation window).

### Parameters

The `OPCConnectServer` function requires three parameters:

`OPCConnectServer(<Machine>, <Server>, <ServerAlias>)`

The parameters are defined in the following table:

| Parameter   | Type   | Description   |
|-------------|--------|---|
| Machine     | String | A reference to the machine on which the server is running (for more information, see “Defining a reference to a machine” on <a href="#">page B.26</a> ).  |
| Server      | String | The name of the server to which you want to open a connection. If you are not sure of a server’s name, you can print the name of all servers running on a particular machine (for more information, see “OPCListItems function” on <a href="#">page B.10</a> ). |
| ServerAlias | String | The alias you want to assign to this server; the alias can then be used to identify the server in other OPC functions.  |

## Return value


An integer value that indicates the success of the function:

- 0 The connection was opened successfully.
- 4 The server name was null. Use the `OPCListItems` function to verify that you are correctly identifying the machine and server name (see “OPCListItems function” on [page B.10](#)). You can also try alternate methods of referencing the machine (see “Defining a reference to a machine” on [page B.26](#)).
- 5 The server refused the connection.

## OPCDisconnect function

### Description

The `OPCDisconnect` function closes an open connection to an OPC server. Closing a connection frees the server resources that are used to maintain the connection. After closing a connection, you can no longer read or write to items on the disconnected server.

All connections to OPC servers close automatically at the end of a model run and when the AutoMod runtime environment is closed (either because you click **File > Exit** or because you click **Close**  in the Simulation window).

---

**Note** If a connection to an OPC server is unexpectedly closed during a model run (for example, because the server shuts down or becomes unresponsive), the model terminates automatically.

---

## Parameters

The `OPCDisconnect` function requires one parameter:

`OPCDisconnect(<Server>)`

The parameter is defined in the following table:

| Parameter | Type   | Description  |
|-----------|--------|--|
| Server    | String | The name or alias of the connected server from which you want to disconnect ( for information on how to define a server's alias, see "OPCConnectServer function" on <a href="#">page B.5</a> ). If this parameter is null, the model disconnects from all OPC servers. |

## Return value

You can ignore the integer return value; this function always returns zero.

# OPCFlushWrite function

## Description

The `OPCFlushWrite` function allows you to verify one or more values that have been written to the OPC server synchronously without validation using the `OPCWriteInteger`, `OPCWriteReal`, or `OPCWriteString` functions.

The `OPCFlushWrite` function verifies written values by comparing the current value on the OPC server with the last value written by the model. Possible reasons that the function *cannot* verify an item's value include:

- Another client application has changed the item's value after it was written by the model, but before the `OPCFlushWrite` function was called to verify the item's value.
- The OPC server changed the written value; for example, by truncating a string or truncating or rounding a real value. The model provides a tolerance for the amount that real values can change and still be verified on the server. The tolerance is determined by the deadband value defined for the item's group. If the following relation is true, the item value is successfully verified on the server:

$$\frac{|\text{value read from server} - \text{value written to server}|}{\text{value read from server}} < \frac{\text{deadband}}{100}$$

For more information about defining a group's deadband, see "Adding groups to a server" on [page 2.16](#).

## Parameters

The `OPCFlushWrite` function requires one parameter:

`OPCFlushWrite(<Item>)`

The parameter is defined in the following table:

| Parameter | Type   | Description  |
|-----------|--------|--|
| Item      | String | A reference to the item whose value you want to verify (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ). If this parameter is null, the model verifies all values that have been written to items on the OPC server but have not yet been verified. |

## Return value

An integer value that indicates the success of the function:

- 0 The item’s value was written successfully.
- 1 The item’s value could not be verified.
- 2 The OPC item was not found. Verify that you are correctly referencing the item; see “Defining a reference to an item” on [page B.26](#).

# OPCInit function

## Description

The `OPCInit` function opens a connection to one or more servers by reading a configuration file. The function uses the file to create groups on the connected server(s) and to add items to each group.

You can create a configuration file using the OPC Utility. For more information, see “Opening connections using an OPC Utility configuration file” on [page 2.9](#) and the OPC Utility online help.

## Parameters

The `OPCInit` function requires one parameter:

`OPCInit(<ConfigFile>)`

The parameter is defined in the following table:

| Parameter  | Type   | Description   |
|------------|--------|---|
| ConfigFile | String | The name of a configuration file; you can specify a path preceding the configuration file name (for example, C:\models\test\testmodel.dat). If the name does not include a path, the model looks for the configuration file in the model directory. |

### Return value

An integer value that indicates the success of the function:

- 0 The configuration file was processed successfully.
- 3 The configuration file could not be found; verify that the configuration file's path and file name are correct.

## OPCListConnections function

### Description

The `OPCListConnections` function allows you to verify the connections that have been opened to OPC servers by printing connection information to the Messages dialog box during a simulation. The function prints the currently connected servers, the groups that have been defined, and the items that have been added to each group. The function also prints each item's properties (see "Item properties" on [page 2.5](#)).

For an example of the `OPCListConnections` function's printed output, see "Example - Printing connection information to the Messages dialog box" on [page 2.24](#).

---

**Note** Some OPC servers do not support this feature; if the function does not list any connections, refer to the documentation that came with your server to determine how to list open connections to the server.

---

### Parameters

The `OPCListConnections` function does not require any parameters.

### Return value

The total number of items that have been added to currently connected OPC servers.

# OPCListItems function

## Description

Before you can call functions to add items to a group on a server, you must know what items are available on the server. The `OPCListItems` function allows you to print a list of all items on an OPC server or, if you are unsure of a server's name, all items on all servers running on a particular machine. The list is printed to the Messages dialog box during a model run.

After running the model, you can print a hardcopy of the item list by printing the `<modelname>.messages` file in the model directory. You can use this printed list when writing functions to connect to servers, define groups, and add items to each group (for more information, see “Opening a connection using functions” on [page 2.12](#)). After using the item list to configure all desired connections, comment out the `OPCListItems` function or remove it from your model.

**Important** Some OPC servers do not support this feature; using the `OPCListItems` function with these servers might cause the servers to malfunction or might cause the model to become unresponsive. In this case, refer to the documentation that came with your server for information about how to list the items on the server.

## Parameters

The `OPCListItems` function requires two parameters:

`OPCListItems(<Machine>,<Server>)`

The parameters are defined in the following table:

| Parameter | Type   | Description  |
|-----------|--------|--|
| Machine   | String | A reference to the machine on which one or more servers are running ( for information on how to define a reference to a machine, see “Defining a reference to a machine” on <a href="#">page B.26</a> ). |
| Server    | String | The name of the server for which you want to list items; if this value is null, the function lists items for all OPC servers on the specified machine.   |

## Return value

You can ignore the integer return value; this function always returns zero.



## OPCReadInteger function

### Description

The `OPCReadInteger` function allows you to read an integer value from an item on an OPC server at any point during a model run.

---

**Note** Before you can read an item's value, you must open a connection to the server and add the item to a group (for more information, see "Opening connections" on [page 2.9](#)).

---

Regardless of the item's type on the server, the function automatically attempts to cast the item's value to an integer value in the model.

After calling the `OPCReadInteger` function, the model automatically sets the value of the `OPCQuality` and `OPCTimestamp` variables to include information about the read value. You can use these variables to check the quality of the data or the time it was received from the server (for more information, see "Reading item values" on [page 2.30](#)).

### Parameters

The `OPCReadInteger` function requires one parameter:

`OPCReadInteger(<Item>)`

The parameter is defined in the following table:

| Parameter | Type   | Description   |
|-----------|--------|---|
| Item      | String | A reference to the item whose value you want to read (for more information, see "Defining a reference to an item" on <a href="#">page B.26</a> ). |

### Return value

The current integer value of the referenced item on the OPC server.

## OPCReadReal function

Description

### Description

The `OPCReadReal` function allows you to read a real value from an item on an OPC server at any point during a model run.

---

**Note** Before you can read an item's value, you must open a connection to the server and add the item to a group (for more information, see "Opening connections" on [page 2.9](#)).

---

Regardless of the item's type on the server, the function automatically attempts to cast the item's value to a real value in the model.

After calling the `OPCReadReal` function, the model automatically sets the value of the `OPCQuality` and `OPCTimestamp` variables to include information about the read value. You can use these variables to check the quality of the data or the time it was received from the server (for more information, see “Reading item values” on [page 2.30](#)).

### Parameters

The `OPCReadReal` function requires one parameter:

`OPCReadReal(<Item>)`

The parameter is defined in the following table:

| Parameter | Type   | Description   |
|-----------|--------|---|
| Item      | String | A reference to the item whose value you want to read (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ). |

### Return value

The current real value of the referenced item on the OPC server.

## OPCReadString function

### Description

The `OPCReadString` function allows you to read a string value from an item on an OPC server at any point during a model run.

**Note** Before you can read an item’s value, you must open a connection to the server and add the item to a group (for more information, see “Opening connections” on [page 2.9](#)).

Regardless of the item’s type on the server, the function automatically attempts to cast the item’s value to a string value in the model.

After calling the `OPCReadString` function, the model automatically sets the value of the `OPCQuality` and `OPCTimestamp` variables to include information about the read value. You can use these variables to check the quality of the data or the time it was received from the server (for more information, see “Reading item values” on [page 2.30](#)).

### Parameters

The `OPCReadString` function requires one parameter:

`OPCReadString(<Item>)`

The parameter is defined in the following table:

| Parameter | Type   | Description   |
|-----------|--------|---|
| Item      | String | A reference to the item whose value you want to read (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ). |

### Return value

The current string value of the referenced item on the OPC server.

## OPCRefreshSubscription function

### Description

The `OPCRefreshSubscription` function causes an OPC server to update the model with the current value of a subscribed item. By passing a `null` item reference to the function, you can cause the connected OPC servers to update the model with the current value of all subscribed items.

If a subscribed item has been associated with an AutoMod variable, the variable’s value is updated to the current item value on the server. In addition, the `opccallback` function is called for each subscribed item that is refreshed.

For more information about refreshing subscribed items, see “Refreshing item subscriptions” on [page 2.35](#).

### Parameters

The `OPCRefreshSubscription` function requires one parameter:

`OPCRefreshSubscription(<Item>)`

The parameter is defined in the following table:

| Parameter | Type   | Description  |
|-----------|--------|--|
| Item      | String | A reference to the item whose value you want to refresh (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ) or the name of a group. If you specify a group name, the model refreshes the subscription of every item in the group. If the parameter is null, the function refreshes the value of all subscribed items in the model. |

## Return value

An integer value that indicates the success of the function:

- 0 The subscription(s) were refreshed successfully.
- 1 The referenced item was invalid. Verify that you are correctly referencing the item (see “Defining a reference to an item” on [page B.26](#)) and that the connection to the item’s server is open (see “Verifying connections” on [page 2.22](#)).

# OPCSubscribe function

## Description

The `OPCSubscribe` function allows you to subscribe to an individual item or to a group of items on an OPC server. After you have subscribed to an item, the OPC server updates the model whenever the item’s value changes on the server by a percentage that is greater than the deadband defined for the item’s group (for more information, see “Group properties” on [page 2.6](#)).

Immediately after you call the `OPCSubscribe` function, the model attempts to initialize the subscribed item by reading the item’s current value on the server. The model then sets the value of the item’s associated AutoMod variable (if defined) and calls the `opccallback` function defined for the item (if defined).

**Important** In some cases, the model might not successfully initialize subscribed item values due to the latency of the OPC server. To ensure that all item values are successfully synchronized with the server, call the `OPCRefreshSubscription` function after subscribing to the appropriate items. For more information, see “`OPCRefreshSubscription` function” on [page B.13](#).

When an OPC server updates the model because a subscribed item's value has changed, the model responds in the following ways:

- If an AutoMod variable has been associated with the subscribed item (that is, the value of the second parameter in the `OPCSubscribe` function is *not null*) and the quality of the item value is “Good” (192 or above), the variable's value is automatically set to the current item value received from the server.
- If one or more `opccallback` functions have been defined for the item, the model calls the `opccallback` function with the highest precedent (for more information, see “`opccallback` function” on [page B.23](#)).

For more information about item subscription, see “Subscribing to groups and items” on [page 2.33](#).

## Parameters

The `OPCSubscribe` function requires two parameters:

`OPCSubscribe(<Item>, <Variable>)`

The parameters are defined in the following table:

| Parameter | Type   | Description   |
|-----------|--------|---|
| Item      | String | A reference to the item to which you want to subscribe (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ) or the name of a group. If you specify a group name, the model subscribes to every item in the group; in this case, the Variable parameter must be null.<br><br><b>Note</b> If the same group name is defined on more than one server, the software subscribes to all items in the group on each connected server. |
| Variable  | String | The name of an AutoMod variable that you want to associate with the subscribed item; if this value is null, no variable is associated with the item and an <code>opccallback</code> function must be defined to respond to changes in the subscribed item's value.  |

## Return value

An integer value that indicates the success of the function:

- 0 The subscription is registered.
- 2 The OPC item was not found. Verify that you are correctly referencing the item; see “Defining a reference to an item” on [page B.26](#).
- 5 The OPC server returned an error.

## OPCWriteInteger function

### Description

The `OPCWriteInteger` function sets the integer value of an item on an OPC server. When you use the `OPCWriteInteger` function, the model does not automatically verify that the item value was successfully written to the server; consequently, after writing one or more values, call the `OPCFlushWrite` function to verify that the server was successfully updated (see “Writing item values synchronously without validation” on [page 2.28](#)). If the value you are writing is not of type `Integer`, the model attempts to cast the value to the item’s type.

---

**Tip** The `OPCWriteInteger` function is useful for improving model performance when you are writing a large number of values to the OPC server and want to wait until all values have been written to verify that the item’s were successfully updated. If you are writing only a few values, use the `OPCWriteIntegerWait` function instead (see “`OPCWriteIntegerWait` function” on [page B.17](#)).

---

**Important** For a model to write to an item’s value, the item must be defined on the OPC server with writable access rights. You can print the access rights of all items the model has added to groups on the server (see “Printing connection information to the Messages dialog box” on [page 2.23](#)). If an item value is not writable, any attempt by a model to write the value is ignored. For information about changing item access rights, see the documentation that came with your OPC server.

### Parameters

The `OPCWriteInteger` function requires two parameters:

`OPCWriteInteger(<Item>, <Value>)`

The parameters are defined in the following table:

| Parameter | Type    | Description  |
|-----------|---------|--|
| Item      | String  | A reference to the item whose value you want to write (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ). |
| value     | Integer | The value you are writing to the item.   |

### Return value

An integer value that indicates the success of the function:

- 0 The OPC server did not return an error.
- 2 The OPC item was not found. Verify that you are correctly referencing the item; see “Defining a reference to an item” on [page B.26](#).
- 5 The OPC server returned an error.

## OPCWriteIntegerWait function

### Description

The `OPCWriteIntegerWait` function sets the integer value of an item on an OPC server, and then compares the server’s item value to the written value to verify that the write was successful. If the value you are writing is not of type `Integer`, the model attempts to cast the value to the item’s type.

**Important** For a model to write to an item’s value, the item must be defined on the OPC server with writable access rights. You can print the access rights of all items the model has added to groups on the server (see “Printing connection information to the Messages dialog box” on [page 2.23](#)). If an item value is not writable, any attempt by a model to write the value is ignored. For information about changing item access rights, see the documentation that came with your OPC server.

### Parameters

The `OPCWriteIntegerWait` function requires three parameters:

`OPCWriteIntegerWait(<Item>, <Value>, <Timeout>)`

The parameters are defined in the following table:

| Parameter | Type    | Description  |
|-----------|---------|--|
| Item      | String  | A reference to the item whose value you want to write (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ). |
| Value     | Integer | The value you are writing to the item.   |
| Timeout   | Real    | The amount of time, in seconds, the model waits to verify that the item is written to the server before continuing processing.                     |

### Return value

An integer value that indicates the success of the function:

- 0 The item’s value was written successfully.
- 2 The OPC item was not found. Verify that you are correctly referencing the item; see “Defining a reference to an item” on [page B.26](#).
- 5 The OPC server returned an error.
- 7 The server was unable to verify that the data was written and received.
- 8 The server timed out before acknowledging that the data was received.

## OPCWriteReal function

### Description

The `OPCWriteReal` function sets the real value of an item on an OPC server. When you use the `OPCWriteReal` function, the model does not automatically verify that the item value was successfully written to the server; consequently, after writing one or more values, call the `OPCFlushWrite` function to verify that the server was successfully



updated (see “Writing item values synchronously without validation” on [page 2.28](#)). If the value you are writing is not of type `Real`, the model attempts to cast the value to the item’s type.

---

**Tip** The `OPCWriteReal` function is useful for improving model performance when you are writing a large number of values to the OPC server and want to wait until all values have been written to verify that the item’s were successfully updated. If you are writing only a few values, use the `OPCWriteRealWait` function instead (see “`OPCWriteRealWait` function” on [page B.20](#)).

---

**Important** For a model to write to an item’s value, the item must be defined on the OPC server with writable access rights. You can print the access rights of all items the model has added to groups on the server (see “Printing connection information to the Messages dialog box” on [page 2.23](#)). If an item value is not writable, any attempt by a model to write the value is ignored. For information about changing item access rights, see the documentation that came with your OPC server.

## Parameters

The `OPCWriteInteger` function requires two parameters:

`OPCWriteReal(<Item>, <Value>)`

The parameters are defined in the following table

| Parameter | Type   | Description  |
|-----------|--------|--|
| Item      | String | A reference to the item whose value you want to write (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ). |
| value     | Real   | The value you are writing to the item.   |

## Return value

An integer value that indicates the success of the function:

- 0 The item’s value was written successfully.
- 2 The OPC item was not found. Verify that you are correctly referencing the item; see “Defining a reference to an item” on [page B.26](#).
- 5 The OPC server returned an error.

# OPCWriteRealWait function

## Description

The `OPCWriteRealWait` function sets the integer value of an item on an OPC server, and then compares the server's item value to the written value to verify that the write was successful. Because the model verifies the write by comparison, any changes that the server makes to the written value can cause the model to timeout. For example, if the OPC server rounds or truncates the real value, resulting in a change in the value that exceeds the value's tolerance, the write will timeout. Each real value's tolerance is determined by the deadband defined for the item's group. If the following relation is true, the model assumes the item value has been written to the server and does *not* timeout:

$$\frac{|\text{value read from server} - \text{value written to server}|}{\text{value read from server}} < \frac{\text{deadband}}{100}$$

For more information about defining a group's deadband, see "Adding groups to a server" on [page 2.16](#).

If the value you are writing is not of type `Real`, the model attempts to cast the value to the item's type.

**Important** For a model to write to an item's value, the item must be defined on the OPC server with writable access rights. You can print the access rights of all items the model has added to groups on the server (see "Printing connection information to the Messages dialog box" on [page 2.23](#)). If an item value is not writable, any attempt by a model to write the value is ignored. For information about changing item access rights, see the documentation that came with your OPC server.

## Parameters

The `OPCWriteRealWait` function requires three parameters:

`OPCWriteRealWait(<Item>, <Value>, <Timeout>)`

The parameters are defined in the following table:

| Parameter | Type   | Description  |
|-----------|--------|--|
| Item      | String | A reference to the item whose value you want to write (for more information, see "Defining a reference to an item" on <a href="#">page B.26</a> ). |
| Value     | Real   | The value you are writing to the item.   |
| Timeout   | Real   | The amount of time, in seconds, the model waits to verify that the item is written to the server before continuing processing.                     |

## Return value

An integer value that indicates the success of the function:

- 0 The item's value was written successfully.
- 2 The OPC item was not found. Verify that you are correctly referencing the item; see “Defining a reference to an item” on [page B.26](#).
- 5 The OPC server returned an error.
- 7 The server was unable to verify that the data was written and received.
- 8 The server timed out before acknowledging that the data was received.

## OPCWriteString function

### Description

The `OPCWriteString` function sets the string value of an item on an OPC server. When you use the `OPCWriteString` function, the model does not automatically verify that the item value was successfully written to the server; consequently, after writing one or more values, call the `OPCFlushWrite` function to verify that the server was successfully updated (see “Writing item values synchronously without validation” on [page 2.28](#)). If the value you are writing is not of type `String`, the model attempts to cast the value to the item's type.

---

**Tip** The `OPCWriteString` function is useful for improving model performance when you are writing a large number of values to the OPC server and want to wait until all values have been written to verify that the item's were successfully updated. If you are writing only a few values, use the `OPCWriteStringWait` function instead (see “`OPCWriteStringWait` function” on [page B.22](#)).

---

**Important** For a model to write to an item's value, the item must be defined on the OPC server with writable access rights. You can print the access rights of all items the model has added to groups on the server (see “Printing connection information to the Messages dialog box” on [page 2.23](#)). If an item value is not writable, any attempt by a model to write the value is ignored. For information about changing item access rights, see the documentation that came with your OPC server.

### Parameters

The `OPCWriteString` function requires two parameters:

`OPCWriteString(<Item>, <Value>)`

The parameters are defined in the following table:

| Parameter | Type   | Description  |
|-----------|--------|--|
| Item      | String | A reference to the item whose value you want to write (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ). |
| value     | String | The value you are writing to the item.   |

## Return value

An integer value that indicates the success of the function:

- 0 The item’s value was written successfully.
- 2 The OPC item was not found. Verify that you are correctly referencing the item; see “Defining a reference to an item” on [page B.26](#).
- 5 The OPC server returned an error.

# OPCWriteStringWait function

## Description

The `OPCWriteStringWait` function sets the string value of an item on an OPC server, and then compares the server’s item value to the written value to verify that the write was successful. Because the model verifies the write by comparison, any changes that the server makes to the written value can cause the model to timeout. For example, if you are writing a string value and the OPC server truncates the string, the compared values will not match and the write will timeout. If the value you are writing is not of type `String`, the model attempts to cast the value to the item’s type.

**Important** For a model to write to an item’s value, the item must be defined on the OPC server with writable access rights. You can print the access rights of all items the model has added to groups on the server (see “Printing connection information to the Messages dialog box” on [page 2.23](#)). If an item value is not writable, any attempt by a model to write the value is ignored. For information about changing item access rights, see the documentation that came with your OPC server.

## Parameters

The `OPCWriteStringWait` function requires three parameters:

`OPCWriteStringWait(<Item>, <Value>, <Timeout>)`

The parameters are defined in the following table:

| Parameter | Type   | Description  |
|-----------|--------|--|
| Item      | String | A reference to the item whose value you want to write (for more information, see “Defining a reference to an item” on <a href="#">page B.26</a> ). |
| Value     | String | The value you are writing to the item.   |
| Timeout   | Real   | The amount of time, in seconds, the model waits to verify that the item is written to the server before continuing processing.                     |

### Return value

An integer value that indicates the success of the function:

- 0 The item’s value was written successfully.
- 2 The OPC item was not found. Verify that you are correctly referencing the item; see “Defining a reference to an item” on [page B.26](#).
- 5 The OPC server returned an error.
- 7 The server was unable to verify that the data was written and received.
- 8 The server timed out before acknowledging that the data was received.

---

## User-defined OPC functions

AutoMod provides one predeclared OPC function that you can define in a model. During a model run, the software calls this function automatically and passes the required parameters to the function; you can use these parameters to make logical decisions in your function code.

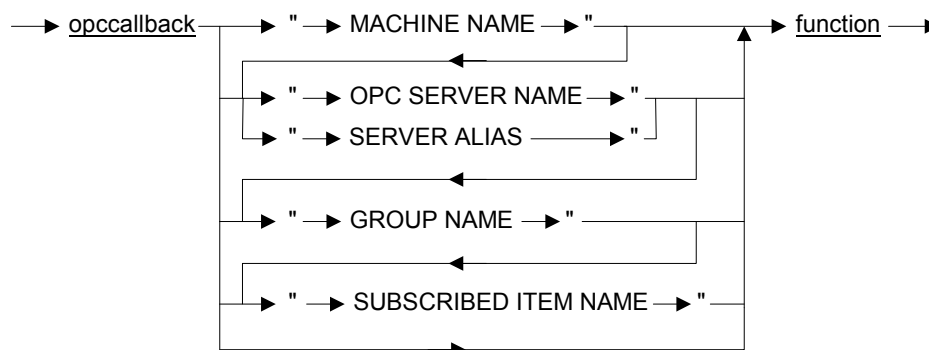
### opccallback function

#### Description

An `opccallback` function is a user-defined function that a model calls when the value of a subscribed item on the server changes by a percentage that is greater than the deadband defined for the item’s group. By defining one or more `opccallback` functions in a model, you can respond to changes in subscribed item values with custom processing in the model (for more information, see “Responding to changes in subscribed item values” on [page 2.37](#)).

You can define one `opccallback` function to respond to all changes in subscribed item values, and you can define separate `opccallback` functions for a specific machine, server, group, or subscribed item. A syntax diagram for defining each `opccallback` function is shown here:

### SYNTAX




---

**Note** For more information about defining an alias for a server, see “OPCConnectServer function” on [page B.5](#).

---

If more than one `opccallback` function applies to the same item, only one function is called. The software determines which function to call by using the following order of precedence:

1. Function defined for a specific item using the machine, server, group, and item name (highest precedence)
2. Function defined for a specific item using the server alias, group, and item name
3. Function defined for a specific item using the group and item name
4. Function defined for a specific item using only the item name
5. Function defined for a group of items using the server alias and group name
6. Function defined for a group of items using only the group name
7. Function defined for all items on a server using the server alias
8. Function defined for all items on a server using the server name
9. Function defined for all items on a machine using the machine name
10. Global `opccallback`—no machine, server, group, or item name specified (lowest precedence)

For example, if separate `opccallback` functions are defined for an item’s server, an item’s group, and for the specific item, only the specific item’s function (which has the highest precedence) is called when the item value changes on the server.

## Parameters

The `opccallback` function provides the following system-defined parameters that you can use when writing the function's code:

| Parameter                 | Type    | Description   |
|---------------------------|---------|---|
| <code>theItem</code>      | String  | The name of the item that has changed in value; the machine and server names are not included in the string.  |
| <code>theData</code>      | String  | <p>The current (updated) value of the item on the OPC server.</p> <p>You can cast an item's value from a string to a numeric data type in the model by using the <code>set</code> action to set the value to a variable of the correct type. For example, if the updated item is of type Integer, you can set the value of an AutoMod integer variable to the parameter <code>theData</code>; the string is automatically cast to an integer.</p> |
| <code>theTimestamp</code> | String  | The date and time when the item value changed on the server.  |
| <code>theQuality</code>   | Integer | The quality of the value that was read from the server. For more information on quality codes and their meanings, see Appendix A, "OPC Quality Codes."  |

## Return value

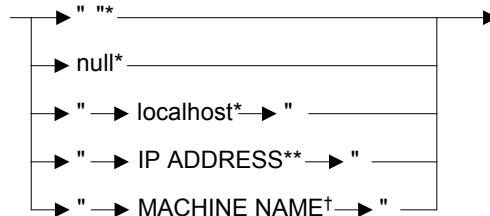
You can return any integer value; the software ignores the returned value.

---

## Defining a reference to a machine

Some OPC functions require a string parameter that refers to a local or networked machine. You can refer to a machine in several ways, which are illustrated in the following syntax diagram:

### MACHINE REFERENCE SYNTAX



\* Refers to the local machine (that is, the machine that is running the model).

\*\* You can specify either an IP loopback address (for example, "127.0.0.1") or an IP address (for example, "10.42.255.60").

† The machine name can either be a short name (for example, "Comp01") or a long name (for example, "Comp01.mycompany.com").

**Important** Depending on the version of Windows you are running and your network configuration, some machine references might fail while others succeed. If you get errors referring to a machine using a short machine name, try a long machine name, an IP address, or one of the other methods.

---

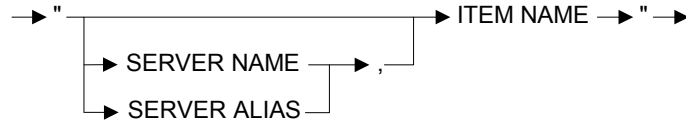
## Defining a reference to an item

Many OPC functions require a string parameter that refers to an item on an OPC server. In general, you can refer to an item simply by using the item's name (this option results in the best model performance). However, if you are connected to multiple OPC servers and there are items on different servers with the same name, you must indicate



which item you are referencing by including a server name or alias in the item reference. The following syntax diagram illustrates the valid syntax for referring to an item:

#### ITEM REFERENCE SYNTAX\*



\* You only need to include a server name or alias in the item reference when you are referring to an item that has the same name as another item on a connected OPC server. Where possible, you can improve model performance by including an item name alone.

---

**Note** When including the optional server name or alias in an item reference, do not put any spaces after the comma in the reference.

---





# Socket Communication Functions

This appendix contains descriptions of the functions used to manage socket connections and send and receive string or C structure messages through socket connections during a simulation.

For a description of the functions used to communicate between synchronized models and the MMS Server, see appendix D, “Multi-Model Synchronization Functions.”

---

**Note** To use the functions listed in this appendix, you must purchase and authorize either the Model Communications module or the Model Communications Plus module.

---

**Important** To use model communications functions in custom C code, you must include the `user.h` header file; type the following at the beginning of each C source file:

```
#include "user.h"
```

---

## System functions

System functions are predefined functions that you can call in a model to manage connections and send or read string or C structure messages during a simulation. Each function’s explanation includes the function’s name and required parameters, a description of how to use the function, and the function’s return value.

### AcceptSocketPort function

#### Description

Use the `AcceptSocketPort` function to send a connection acceptance to a client machine. The port is identified by a port number greater than 1000. For more information, see “Example 1 – Accepting a connection using a port number” on [page 3.10](#).

**Important** The port numbers that identify the client connection attempt and server acceptance must be the same.

## Parameters

The `AcceptSocketPort` function requires one parameter:

`AcceptSocketPort(<Port_number>)`

The parameter is defined in the following table:

| Parameter   | Type    | Description                               |
|-------------|---------|---|
| Port_number | Integer | A port number (must be greater than 1000) |

## Return value

A socket pointer that points to the open connection.

# AcceptSocketService function

## Description

Use the `AcceptSocketService` function to send a connection acceptance to a client machine. The port is identified by a service name (see “Defining a service name” on [page 3.6](#)). For more information, see “Example 2 – Accepting a connection using a service name” on [page 3.11](#).

**!** **Important** The port number referred to by the service name must be the same for both the client connection attempt and the server acceptance.

## Parameters

The `AcceptSocketService` function requires one parameter:

`AcceptSocketService(<Service>)`

The parameter is defined in the following table:

| Parameter | Type   | Description  |
|-----------|--------|--|
| Service   | String | A service name (string literals must be in quotes) |

## Return value

A socket pointer that points to the open connection.

## CloseSocket function

### Description

Use the `CloseSocket` function to manually close a socket connection during a simulation. Socket connections are closed automatically at the end of a simulation. Connections are also automatically closed if there is an error when reading or sending a message.

### Parameters

The `CloseSocket` function requires one parameter:

`CloseSocket(<Socket>)`

The parameter is defined in the following table:

| Parameter | Type      | Description                                   |
|-----------|-----------|---|
| Socket    | SocketPtr | A pointer to the connection you want to close |

### Return value

An integer value that indicates the status of the socket connection:

- 0 The socket connection did not close
- 1 The socket connection is closed

## ConnectSocketPort function

### Description

Use the `ConnectSocketPort` function to send a connection attempt to a server machine. The port is identified by a port number greater than 1000. For more information, see “Example 1 – Attempting a connection using a port number” on [page 3.9](#).

**! Important** The port numbers that identify the client connection attempt and server acceptance must be the same.

### Parameters

The `ConnectSocketPort` function requires two parameters:

`ConnectSocketPort(<Server>, <Port_number>)`

The parameters are defined in the following table:

| Parameter   | Type    | Description   |
|-------------|---------|---|
| Server      | String  | The server machine's name (string literals must be in quotes) |
| Port_number | Integer | A port number (must be greater than 1000)                     |

### Return value

A socket pointer that points to the open connection.

## ConnectSocketService function

### Description

Use the `ConnectSocketService` function to send a connection attempt to a server machine. The port is identified by a service name (see “Defining a service name” on [page 3.6](#)). For more information, see “Example 2 – Attempting a connection using a service name” on [page 3.9](#).

**Important** The port number referred to by the service name must be the same for both the client connection attempt and the server acceptance.

### Parameters

The `ConnectSocketService` function requires two parameters:

`ConnectSocketService(<Server>, <Service>)`

The parameters are defined in the following table:

| Parameter | Type   | Description   |
|-----------|--------|---|
| Server    | String | The server machine's name (string literals must be in quotes) |
| Service   | String | A service name (string literals must be in quotes)            |

### Return value

A socket pointer that points to the open connection.

## FEvFirstTime function

### Description

Use the `FEvFirstTime` function in the `model nextclock` function to determine the time of the next future event. See “Understanding event-driven simulation” on [page 4.2](#) for a description of how to use this function.

### Parameters

The `FEvFirstTime` function has no parameters.

### Return value

The time of a model’s next future event.

## GetSocketName function

### Description

Use the `GetSocketName` function to find the name of a machine to which you have opened a socket connection and the random number assigned to that connection. The same information that is returned by this function can also be obtained by printing a socket pointer to the Message window or to a file.

### Parameters

The `GetSocketName` function requires one parameter:

`GetSocketName(<Socket>)`

The parameter is defined in the following table:

| Parameter | Type      | Description                                       |
|-----------|-----------|---|
| Socket    | SocketPtr | The socket about which you would like information |

### Return value

A string that contains the socket number and the name of the connected machine (for example, `Socket 164 to host.example.com`).

## GetSocketHost function

### Description

Use the `GetSocketHost` function to find the name of a machine to which you have opened a socket connection.

## Parameters

The `GetSocketHost` function requires one parameter:

`GetSocketHost(<Socket>)`

The parameter is defined in the following table:

| Parameter | Type      | Description                                       |
|-----------|-----------|---|
| Socket    | SocketPtr | The socket about which you would like information |

## Return value

A string that contains the name of the connected machine (for example, `host.example.com`).

# GetSocketNum function

## Description

Use the `GetSocketNum` function to find the number assigned to a socket connection. Socket numbers are randomly assigned and are not used by any other function; they are provided for your information only.

## Parameters

The `GetSocketNum` function requires one parameter:

`GetSocketNum(<Socket>)`

The parameter is defined in the following table:

| Parameter | Type      | Description                                       |
|-----------|-----------|---|
| Socket    | SocketPtr | The socket about which you would like information |

## Return value

The random number assigned to a socket connection.

# MachineName function

## Description

Use the `MachineName` function to obtain the server machine name when both the client and server models are running on the same machine. The server machine name is used with both the `ConnectSocketPort` and `ConnectSocketService` functions. For more information, see “Attempting a connection (client)” on [page 3.8](#).



## Parameters

The `MachineName` function does not require any parameters.

## Return value

A string that is the name of the machine on which the model is running.

# NetworkConvertShort function

## Description

Use the `NetworkConvertShort` function in a model to change the sorting order of a C integer short expression before sending the value to, or after receiving the value from, a machine that uses the network byte order. For more information, see “Converting numeric values in C structure messages to the network byte order” on [page 3.14](#).

## Parameters

The `NetworkConvertShort` function requires one parameter:

`NetworkConvertShort(<Value>)`

The parameter is defined in the following table:

| Parameter | Type  | Description                  |
|-----------|-------|------------------------------|
| value     | Short | A C integer short expression |

## Return value

The short value converted to/from the network standard.

# NetworkConvertInteger function

## Description

Use the `NetworkConvertInteger` function in a model to change the sorting order of a C long integer or AutoMod integer before sending the value to, or after receiving the value from, a machine that uses the network byte order. For more information, see “Converting numeric values in C structure messages to the network byte order” on [page 3.14](#).

## Parameters

The `NetworkConvertInteger` function requires one parameter:

`NetworkConvertInteger(<Value>)`

The parameter is defined in the following table:

| Parameter | Type          | Description                                    |
|-----------|---------------|--|
| Value     | Long, Integer | A C long integer or AutoMod integer expression |

### Return value

The integer/long value converted to/from the network standard.

## NetworkConvertFloat function

### Description

Use the `NetworkConvertFloat` function in a model to change the sorting order of a C float before sending the value to, or after receiving the value from, a machine that uses the network byte order. For more information, see “Converting numeric values in C structure messages to the network byte order” on [page 3.14](#).

### Parameters

The `NetworkConvertFloat` function requires one parameter:

`NetworkConvertFloat(<Value>)`

The parameter is defined in the following table:

| Parameter | Type  | Description          |
|-----------|-------|----------------------|
| value     | Float | A C float expression |

### Return value

The float value converted to/from the network standard.

## NetworkConvertDouble function

### Description

Use the `NetworkConvertDouble` function in a model to change the sorting order of a C double or AutoMod real, time, velocity, distance, or acceleration expression before sending the value to, or after receiving the value from, a machine that uses the network byte order. For more information, see “Converting numeric values in C structure messages to the network byte order” on [page 3.14](#).

### Parameters

The `NetworkConvertDouble` function requires one parameter:

`NetworkConvertDouble(<Value>)`

The parameter is defined in the following table:

| Parameter | Type   | Description  |
|-----------|--------|--|
| Value     | Double | A C double or AutoMod real, time, velocity, distance, or acceleration expression |

### Return value

The double value converted to/from the network standard.

## ReadSocket function

### Description

Use the `ReadSocket` function to read a C structure message from an open socket connection. For more information, see “Reading a C structure message” on [page 3.21](#).

### Parameters

The `ReadSocket` function requires three parameters:

`ReadSocket(<Socket>, <Msg_structure>, <Msg_size>)`

The parameters are defined in the following table:

| Parameter     | Type      | Description  |
|---------------|-----------|--|
| Socket        | SocketPtr | Points to the connection from which you want to read |
| Msg_structure | String    | The message structure converted to a string          |
| Msg_size      | Integer   | The size of the message in bytes                     |

### Return value

An integer value that indicates the length of the read message in bytes. If the return value is zero, it indicates that no message was read.

## ReadSocketString function

### Description

Use the `ReadSocketString` function to read a string message from an open socket connection. For more information, see “Reading a string message” on [page 3.20](#).

## Parameters

The `ReadSocketString` function requires one parameter:

`ReadSocketString(<Socket>)`

The parameter is defined in the following table:

| Parameter | Type      | Description  |
|-----------|-----------|--|
| Socket    | SocketPtr | Points to the connection from which you want to read |

## Return value

The string message that is read from the socket connection.

# SendSocket function

## Description

Use the `SendSocket` function to send a C structure message through an open socket connection. For more information, see “Sending a C structure message” on [page 3.17](#).

## Parameters

The `SendSocket` function requires three parameters:

`SendSocket(<Socket>, <Msg_structure>, <Msg_size>)`

The parameters are defined in the following table:

| Parameter     | Type      | Description   |
|---------------|-----------|---|
| Socket        | SocketPtr | Points to the connection through which you want to send a message |
| Msg_structure | String    | The message structure converted to a string                       |
| Msg_size      | Integer   | The size of the message in bytes                                  |

## Return value

An integer value that indicates the length of the sent message in bytes. If the return value is zero, it indicates that the message was not sent.

## SendSocketString function

### Description

Use the `SendSocketString` function to send a string message through an open socket connection. For more information, see “Sending a string message” on [page 3.16](#).

### Parameters

The `SendSocketString` function requires three parameters:

`SendSocketString(<Socket>, <Message>)`

The parameters are defined in the following table:

| Parameter | Type      | Description   |
|-----------|-----------|---|
| Socket    | SocketPtr | Points to the connection through which you want to send a message |
| Message   | String    | The string message you want to send                               |

### Return value

An integer value that indicates the length of the sent message in bytes. If the return value is zero, it indicates that the message was not sent.

## SetAcceptBlocking function

### Description

Use the `SetAcceptBlocking` function to set a server’s connection acceptance to blocking or timeout. When a server is blocking, a socket that tries to accept a connection will accept repeatedly until a connection with a client is opened. When a server is set to timeout, a socket that tries to accept a connection will repeat for the amount of time defined by the `SetAcceptTimeOut` function. For more information, see “Example 2 – Defining blocking connections” on [page 3.5](#).

### Parameters

The `SetAcceptBlocking` function requires one parameter:

`SetAcceptBlocking(<Toggle>)`

The parameter is defined in the following table:

| Parameter | Type    | Description   |
|-----------|---------|---|
| Toggle    | Integer | Sets connection acceptance to blocking or timeout: <ul style="list-style-type: none"><li>• <b>0</b> - Timeout</li><li>• <b>1</b> - Blocking</li></ul> |

### Return value

An integer value that indicates the status of a server's connection acceptance:

0 Timeout

1 Blocking

## SetAcceptTimeOut function

### Description

Use the `SetAcceptTimeOut` function to set the amount of time a server tries to accept a connection when the server is set to timeout rather than blocking. If a connection is not opened within the defined time period, the server stops accepting connections. For more information, see “Example 1 – Defining timeout connections” on [page 3.5](#).

### Parameters

The `SetAcceptTimeOut` function requires one parameter:

`SetAcceptTimeOut(<Time>)`

The parameter is defined in the following table:

| Parameter | Type    | Description   |
|-----------|---------|---|
| Time      | Integer | The amount of time in seconds that a server attempts to accept a connection |

### Return value

An integer value that indicates the success of the function:

- 0 The timeout value was not set
- 1 The timeout value was set successfully

## SetConnectBlocking function

### Description

Use the `SetConnectBlocking` function to set a client's connection attempts to blocking or timeout. When a client is blocking, a socket that attempts a connection will attempt repeatedly until a connection with a server is opened. When a client is set to timeout, a socket that attempts a connection will repeat for the amount of time defined by the `SetConnectTimeOut` function. For more information, see “Example 2 – Defining blocking connections” on [page 3.5](#).

### Parameters

The `SetConnectBlocking` function requires one parameter:

`SetConnectBlocking(<Toggle>)`

The parameter is defined in the following table:

| Parameter | Type    | Description   |
|-----------|---------|---|
| Toggle    | Integer | Sets connection attempts to blocking or timeout: <ul style="list-style-type: none"><li>• <b>0</b> - Timeout</li><li>• <b>1</b> - Blocking</li></ul> |

### Return value

An integer value that indicates the status of a client's connection attempts:

- 0 Timeout
- 1 Blocking

## SetConnectTimeOut function

### Description

Use the `SetConnectTimeOut` function to set the amount of time a client attempts to open a connection when the client machine is set to timeout rather than blocking. If a connection is not opened within the defined time period, the client stops attempting to open a connection. For more information, see “Example 1 – Defining timeout connections” on [page 3.5](#).

### Parameters

The `SetConnectTimeOut` function requires one parameter:

`SetConnectTimeOut(<Time>)`

The parameter is defined in the following table:

| Parameter | Type    | Description   |
|-----------|---------|---|
| Time      | Integer | The amount of time in seconds that a client attempts to open a connection |

### Return value

An integer value that indicates the success of the function:

- 0 The timeout value was not set
- 1 The timeout value was set successfully

## SetNumAcceptsPort function

### Description

By default, the maximum number of socket connections a port can accept is one. Use the `SetNumAcceptsPort` function to set the total number of connections a port can accept throughout the simulation to any number; use zero to accept an infinite number of connections. The port is identified by a port number greater than 1000. For more information, see “Defining a port’s number of accepted connections” on [page 3.8](#).

### Parameters

The `SetNumAcceptsPort` function requires two parameters:

`SetNumAcceptsPort(<Port_number>,<Connections>)`



The parameters are defined in the following table:

| Parameter   | Type    | Description  |
|-------------|---------|--|
| Port_number | Integer | A port number (must be greater than 1000)  |
| Connections | Integer | The number of connections the port can accept: <ul style="list-style-type: none"><li>• <b>0</b> - infinite</li></ul> |

### Return value

You can ignore the integer return value; this function always returns zero.

## SetNumAcceptsService function

### Description

By default, the maximum number of socket connections a port can accept is one. Use the `SetNumAcceptsService` function to set the number of connections a port can accept to any number; use zero to accept an infinite number of connections. The port is identified by a service name (see “Defining a service name” on [page 3.6](#)). For more information, see “Defining a port’s number of accepted connections” on [page 3.8](#).

### Parameters

The `SetNumAcceptsService` function requires two parameters:

`SetNumAcceptsService(<Service>,<Connections>)`

The parameters are defined in the following table:

| Parameter   | Type    | Description  |
|-------------|---------|--|
| Service     | String  | A service name (string literals must be in quotes)   |
| Connections | Integer | The number of connections the port can accept: <ul style="list-style-type: none"><li>• <b>0</b> - infinite</li></ul> |

### Return value

You can ignore the integer return value; this function always returns zero.

## SetSocketMessages function

### Description

Use the `SetSocketMessages` function to toggle warning messages and other automatic communication messages that are printed to the Message window.

---

**Note** Error messages cannot be turned off.

---

### Parameters

The `SetSocketMessages` function requires one parameter:

`SetSocketMessages(<Toggle>)`

The parameter is defined in the following table:

| Parameter | Type    | Description   |
|-----------|---------|---|
| Toggle    | Integer | Toggles socket messages on or off: <ul style="list-style-type: none"><li>• <b>0</b> - Turn off messages</li><li>• <b>1</b> - Turn on messages</li></ul> |

### Return value

You can ignore the integer return value; this function always returns zero.

## SetSocketNonBlocking function

### Description

Use the `SetSocketNonBlocking` function to set a socket's reading attempts to blocking or timeout. When a socket is blocking, a read attempt will repeat until a message is received. When a socket is set to timeout, a read attempt will repeat for the amount of time defined by the `SetSocketTimeOut` function. If a timeout length has not been defined, a single read attempt is made and then the socket stops trying to read a message. For more information, see "Example – Setting a blocking socket" on [page 3.20](#).

### Parameters

The `SetSocketNonBlocking` function requires two parameters:

`SetSocketNonBlocking(<Socket>, <Toggle>)`

The parameter is defined in the following table:

| Parameter | Type      | Description   |
|-----------|-----------|---|
| Socket    | SocketPtr | Points to the connection through which you want to read a message   |
| Toggle    | Integer   | Sets a socket's reading attempts to blocking or timeout: <ul style="list-style-type: none"><li>• <b>0</b> - Blocking</li><li>• <b>1</b> - Timeout</li></ul> |

### Return value

An integer value that indicates the status of the socket's read attempts:

- 0 Blocking
- 1 Timeout

## SetSocketTimeout function

### Description

Use the `SetSocketTimeout` function to set the amount of time a socket attempts to read a message when the socket is set to timeout rather than blocking. If a message is not read within the defined time period, the socket stops attempting to read a message. For more information, see "Defining the length of reading attempts" on [page 3.20](#).

### Parameters

The `SetSocketTimeout` function requires two parameters:

`SetSocketTimeout(<Socket>, <Time>)`

The parameters are defined in the following table:

| Parameter | Type      | Description  |
|-----------|-----------|--|
| Socket    | SocketPtr | Points to the connection through which you want to read a message      |
| Time      | Integer   | The amount of time in seconds that a socket attempts to read a message |

### Return value

An integer value that indicates the success of the function:

- 0 The timeout value was not set
- 1 The timeout value was set successfully

## ValidSocket function

### Description

Use the `ValidSocket` function to determine if a socket connection is open for reading and sending messages. There are several reasons why a connection may become invalid, for example, because of a network error or because a machine is turned off, a simulation is closed, and so on. For more information, see “Verifying that a socket connection is valid” on [page 3.12](#).

### Parameters

The `ValidSocket` function requires one parameter:

`ValidSocket(<Socket>)`

The parameter is defined in the following table:

| Parameter | Type      | Description  |
|-----------|-----------|--|
| Socket    | SocketPtr | Points to the connection that you want to validate |

### Return value

An integer value that indicates whether or not the socket pointer points to a valid connection:

- 0 The connection is invalid
- 1 The connection is valid

---

## User-defined socket functions

The AutoMod software provides one predeclared socket function that you can define in a model. During a model run, the software calls this function automatically.

## model nextclock function

### Description

The `model nextclock` function is called *after*:

- All current events have been completed

and *before*:

- The simulation clock advances to the next future event

Use the `model nextclock` function with the `FEvFirstTime` function to synchronize two models. For more information, see “Synchronizing two models without the MMS Server” on [page 3.26](#).

### Parameters

The `model nextclock` function has no parameters.

### Return value

You can return any integer value; the software ignores the return value.





# Multi-Model Synchronization Functions

This appendix contains descriptions of the functions used for communicating between synchronized models that are connected to the MMS Server.

For a description of the functions used to establish connections directly between models (without the MMS Server) and send or read string and C structure messages, see Appendix C, “Socket Communication Functions.”

---

**Note** To use model communications functions in custom C code, you must include the “user.h” header file; type the following at the beginning of each C source file:

---

**Important** To use model communications functions in custom C code, you must include the “user.h” header file; type the following at the beginning of each C source file:

```
#include “user.h”
```

---

## System functions

System functions are predefined functions that you can call in a model to communicate between models that are synchronized by the MMS Server. Each function’s explanation includes the function’s name and required parameters, a description of how to use the function, and the function’s return value.

## MMSyncReadMessageInteger function

### Description

Use the `MMSyncReadMessageInteger` function to read an integer value from the message contained in the read message buffer.

Each integer value in the message is contained in an integer field. Call the `MMSyncReadMessageInteger` function once for each integer field from which you want to read a value. For more information, see “Reading an MMS message” on [page 4.13](#).

---

**Note** The `MMSyncReadMessageInteger` function can only be used within the `model mmsyncmessage` function (see above for more information).

---

To determine the number of integer fields contained in the message, use the `MMSyncReadMessageNumIntegers` function (see “`MMSyncReadMessageNumIntegers` function” on [page D.2](#)).

## Parameters

The `MMSyncReadMessageInteger` function requires one parameter:

`MMSyncReadMessageInteger(<Index>)`

The parameter is defined in the following table:

| Parameter | Type    | Description  |
|-----------|---------|--|
| Index     | Integer | The index of an integer field in the read message buffer |

## Return value

The value of an integer field in the read message buffer.

# MMSyncReadMessageNumIntegers function

## Description

Use the `MMSyncReadMessageNumIntegers` function to determine the number of integer fields in the message contained in the read message buffer (for more information, see “Reading an MMS message” on [page 4.13](#)).

---

**Note** The `MMSyncReadMessageNumIntegers` function can only be used within the `model mmsyncmessage` function (for more information, see “`model mmsyncmessage` function” on [page D.18](#)).

---

To read the value of an integer field, pass the index of the field you want to read to the `MMSyncReadMessageInteger` function (for more information, see “`MMSyncReadMessageInteger` function” on [page D.1](#)).

## Parameters

The `MMSyncReadMessageNumIntegers` function has no parameters.



## Return value

The number of integer fields in the message contained in the read message buffer.

# MMSyncReadMessageNumReals function

## Description

Use the `MMSyncReadMessageNumReals` function to determine the number of real fields in the message contained in the read message buffer (for more information, see “Reading an MMS message” on [page 4.13](#)).

---

**Note** The `MMSyncReadMessageNumReals` function can only be used within the `model mmsyncmessage` function (for more information, see “model `mmsyncmessage` function” on [page D.18](#)).

---

To read the value of a real field, pass the index of the field you want to read to the `MMSyncReadMessageReal` function (see “`MMSyncReadMessageReal` function” on [page D.4](#)).

## Parameters

The `MMSyncReadMessageNumReals` function has no parameters.

## Return value

The number of real fields in the message contained in the read message buffer.

# MMSyncReadMessageNumStrings function

## Description

Use the `MMSyncReadMessageNumStrings` function to determine the number of string fields in the message contained in the read message buffer (for more information, see “Reading an MMS message” on [page 4.13](#)).

---

**Note** The `MMSyncReadMessageNumStrings` function can only be used within the `model mmsyncmessage` function (for more information, see “model `mmsyncmessage` function” on [page D.18](#)).

---

To read the value of a string field, pass the index of the field you want to read to the `MMSyncReadMessageString` function (see “`MMSyncReadMessageString` function” on [page D.5](#)).

## Parameters

The `MMSyncReadMessageNumStrings` function has no parameters.

### Return value

The number of string fields in the message contained in the read message buffer.

## MMSyncReadMessagePrint function

### Description

Use the `MMSyncReadMessagePrint` function to print the current content of the read message buffer to a string expression. If the trace level of a model or the MMS Server is equal to two, this function is automatically called before the `model mmsyncmessage` function, and the return value is written to a trace file. For more information, see “Using trace files” on [page 4.25](#).

---

**Note** The `MMSyncReadMessagePrint` function can only be used within the `model mmsyncmessage` function (for more information, see “`model mmsyncmessage` function” on [page D.18](#)).

---

### Parameters

The `MMSyncReadMessagePrint` function has no parameters.

### Return value

A tab-delimited string containing the following information:

- The time the current message was received.
- The message type.
- The values of integer fields in the read message buffer.
- The values of real fields in the read message buffer.
- The values of string fields in the read message buffer.

## MMSyncReadMessageReal function

### Description

Use the `MMSyncReadMessageReal` function to read a real value from the message contained in the read message buffer.

Each real value in the message is contained in a real field. Call the `MMSyncReadMessageReal` function once for each real field from which you want to read a value. For more information, see “Reading an MMS message” on [page 4.13](#).

---

**Note** The `MMSyncReadMessageReal` function can only be used within the `model mmsyncmessage` function (for more information, see “`model mmsyncmessage` function” on [page D.18](#)).

---

To determine the number of real fields contained in the message, use the `MMSyncReadMessageNumReals` function (see “`MMSyncReadMessageNumReals` function” on [page D.3](#)).

## Parameters

The `MMSyncReadMessageReal` function requires one parameter:

`MMSyncReadMessageReal(<Index>)`

The parameter is defined in the following table:

| Parameter | Type    | Description  |
|-----------|---------|--|
| Index     | Integer | The index of a real field in the read message buffer |

## Return value

The value of a real field in the read message buffer.

# MMSyncReadMessageString function

## Description

Use the `MMSyncReadMessageString` function to read a string value from the message contained in the read message buffer.

Each string value in the message is contained in a string field. Call the `MMSyncReadMessageString` function once for each string field from which you want to read a value. For more information, see “Reading an MMS message” on [page 4.13](#).

---

**Note** The `MMSyncReadMessageString` function can only be used within the `model mmsyncmessage` function (for more information, see “`model mmsyncmessage` function” on [page D.18](#)).

---

To determine the number of string fields contained in the message, use the `MMSyncReadMessageNumStrings` function (see “`MMSyncReadMessageNumStrings` function” on [page D.3](#)).

## Parameters

The `MMSyncReadMessageString` function requires one parameter:

`MMSyncReadMessageString(<Index>)`

The parameter is defined in the following table:

| Parameter | Type    | Description  |
|-----------|---------|--|
| Index     | Integer | The index of a string field in the read message buffer |

## Return value

The value of a string field in the read message buffer.

# MMSyncReadMessageStringLength function

## Description

Use the `MMSyncReadMessageStringLength` function to read the length of string fields from the message contained in the read message buffer.

---

**Note** The `MMSyncReadMessageStringLength` function can only be used within the `model mmsyncmessage` function (for more information, see “[model mmsyncmessage function](#)” on [page D.18](#)).

---

To read the number of string fields in the message, use the `MMSyncReadMessageNumStrings` function (for more information, see “[MMSyncReadMessageNumStrings function](#)” on [page D.3](#)). To read the value of the string fields, use the `MMSyncReadMessageString` function (see “[MMSyncReadMessageString function](#)” on [page D.5](#)).

## Parameters

The `MMSyncReadMessageStringLength` function has no parameters.

## Return value

The length of the string fields in the message contained in the read message buffer.

## MMSyncReadMessageTime function

### Description

Use the `MMSyncReadMessageTime` function to retrieve the time (in seconds) when the message currently contained in the read message buffer was sent (see “Reading the time that an MMS message was sent” on [page 4.15](#)).

---

**Note** The `MMSyncReadMessageTime` function can only be used within the `model mmsyncmessage` function (for more information, see “`model mmsyncmessage` function” on [page D.18](#)).

---

### Parameters

The `MMSyncReadMessageTime` function has no parameters.

### Return value

The time value (in seconds) when the received message was sent.

## MMSyncReadMessageType function

### Description

Use the `MMSyncReadMessageType` function to read the type of the message contained in the read message buffer. Using MMS message types, you can anticipate the structure of a message, making it easier both to read and to interpret. For more information, see “Defining the type of an MMS message” on [page 4.10](#).

---

**Note** The `MMSyncReadMessageType` function can only be used within the `model mmsyncmessage` function (for more information, see “`model mmsyncmessage` function” on [page D.18](#)).

---

### Parameters

The `MMSyncReadMessageType` function has no parameters.

### Return value

The type of the message contained in the read message buffer.

## MMSyncSendMessage function

### Description

Use the `MMSyncSendMessage` function to send the message contained in the send message buffer to the MMS Server; the sent message is then broadcast by the server to one or all of the other synchronized models. For more information, see “Sending an MMS message” on [page 4.7](#).

If the trace level of the MMS Server is greater than zero, then the `MMSyncSendMessagePrint` function is automatically called to print the content of the sent message to the trace file (see “`MMSyncSendMessagePrint` function” on [page D.15](#)).

After the message is sent, the `MMSyncSendMessageClear` function is automatically called to clear the message type and the values of the fields in the send message buffer (see “`MMSyncSendMessageClear` function” on [page D.8](#)).

### Parameters

The `MMSyncSendMessage` function has no parameters.

### Return value

An integer value that indicates the length of the sent message in bytes. A return value of zero indicates that no message was sent.

## MMSyncSendMessageClear function

### Description

Use the `MMSyncSendMessageClear` function to clear the values of the fields in the send message buffer; the value of integer and real fields is changed to 0, the value of string fields is changed to “”, and the message type is reset to 5000. The recipient of the message in the buffer is reset to all models in the simulation. The structure of the message (the number of fields) remains unchanged and can be used again for the next message. This function is called automatically after a message is sent using the `MMSyncSendMessage` function. For more information, see “`MMSyncSendMessage` function” on [page D.8](#).

### Parameters

The `MMSyncSendMessageClear` function has no parameters.

### Return value

An integer value that indicates the success of the function:

- 0 The send message buffer was not cleared.
- 1 The send message buffer was cleared successfully.

## MMSyncSendMessageInteger function

### Description

Use the `MMSyncSendMessageInteger` function to write values to the integer fields of the message contained in the send message buffer. Call the function once for each field to which you want to write a value. You can change the number of integer fields in the send message buffer using the `MMSyncSendMessageNumIntegers` function (for more information, see “[MMSyncSendMessageNumIntegers function](#)” on [page D.12](#)).

### Parameters

The `MMSyncSendMessageInteger` function requires two parameters:

`MMSyncSendMessageInteger(<Index>, <Value>)`

The parameters are defined in the following table:

| Parameter | Type    | Description   |
|-----------|---------|---|
| Index     | Integer | The index of the integer field to which you are writing a value |
| value     | Integer | The value that is written to the integer field                  |

### Return value

An integer value that indicates the success of the function:

- 0 The value was not written to the integer field.
- 1 The value was successfully written to the integer field.

---

**Note** If the return value of the function is zero, check the value of the first parameter to make sure you are specifying a valid integer field in the message. Another reason the function may fail is if the model is not connected to the MMS Server.

---

## MMSyncSendMessageModel function

### Description

Messages that are sent by a model to the MMS Server are broadcast, by default, to each synchronized model in the simulation (including the model from which the message originated). The `MMSyncSendMessageModel` function sets the message in the read message buffer to be broadcast by the MMS Server only to the specified model. This function must be called before the `MMSyncSendMessage` function is called.

---

**Note** You can only specify one recipient model for the message. After the message is sent, the recipient for future messages is reset to all models in the simulation.

---

For more information, see “Defining the recipient of an MMS message” on [page 4.11](#).

### Parameters

The `MMSyncSendMessageModel` function requires one parameter:

`MMSyncSendMessageModel(<Model>)`



The parameter is defined in the following table:

| Parameter | Type   | Description   |
|-----------|--------|---|
| Model     | String | The name of the model to which the next sent message is broadcast; by default, the software calls this function using the string value <code>MMS_ALL</code> , which causes the next sent message to be broadcast to all of the synchronized models. |

### Return value

An integer value that indicates the success of the function:

- 0 The model value was not set; the message in the buffer will be sent to all models in the simulation.
- 1 The model value was set; the message in the buffer will be sent only to the specified model.

## MMSyncSendMessageNumDatas function

### Description

Use the `MMSyncSendMessageNumDatas` function to define the number of fields in the send message buffer for each data type. The function defines the number of integer, real, and string values, and the length of string values, that can be sent in a message from this machine to the MMS Server.

After creating fields in the send message buffer, you can call a function for each field to write a value to that field in the buffer; the function you call depends on the type of field (integer, real, or string). For more information, see “Defining the content of an MMS message” on [page 4.9](#).

### Parameters

The `MMSyncSendMessageNumDatas` function requires four parameters:

```
MMSyncSendMessageNumDatas(<Int_fields>,<Real_fields>,  
                           <String_fields>,<String_length>)
```

The parameters are defined in the following table:

| Parameter     | Type    | Description  |
|---------------|---------|--|
| Int_fields    | Integer | The number of integer fields to create in the send message buffer                                  |
| Real_fields   | Integer | The number of real fields to create in the send message buffer                                     |
| String_fields | Integer | The number of string fields to create in the send message buffer                                   |
| String_length | Integer | The length of each string field (the number of characters that can be stored in each string field) |

### Return value

An integer value that indicates the success of the function:

- 0 The fields were not created in the message buffer.
- 1 The fields were successfully created in the message buffer.

## MMSyncSendMessageNumIntegers function

### Description

Use the `MMSyncSendMessageNumIntegers` function to define the number of integer fields in the send message buffer; this is the number of integer values that can be sent in a message from this machine to the MMS Server. To write a value to an integer field, use the `MMSyncSendMessageInteger` function. For more information, see “`MMSyncSendMessageInteger` function” on [page D.9](#).

**Tip** Instead of calling a separate function to define the number of fields for each data type, you can define the number of integer, real, and string fields in the send message buffer using one function. For more information, see “`MMSyncSendMessageNumIntegers` function” on [page D.11](#).

### Parameters

The `MMSyncSendMessageNumIntegers` function requires one parameter:

`MMSyncSendMessageNumIntegers(<Int_fields>)`

The parameter is defined in the following table:

| Parameter  | Type    | Description   |
|------------|---------|---|
| Int_fields | Integer | The number of integer fields to create in the send message buffer |

### Return value

An integer value that indicates the success of the function:

- 0 The integer fields were not created in the message buffer.
- 1 The integer fields were successfully created in the message buffer.

## MMSyncSendMessageNumReals function

### Description

Use the `MMSyncSendMessageNumReals` function to define the number of real fields in the send message buffer; this is the number of real values that can be sent in a message from this machine to the MMS Server. To write a value to a real field, use the `MMSyncSendMessageReal` function. For more information, see “`MMSyncSendMessageReal` function” on [page D.16](#).

**Tip** Instead of calling a separate function to define the number of fields for each data type, you can define the number of integer, real, and string fields in the send message buffer using one function. For more information, see “`MMSyncSendMessageNumDatas` function” on [page D.11](#).

### Parameters

The `MMSyncSendMessageNumReals` function requires one parameter:

`MMSyncSendMessageNumReals(<Real_fields>)`

The parameter is defined in the following table:

| Parameter   | Type    | Description  |
|-------------|---------|--|
| Real_fields | Integer | The number of real fields to create in the send message buffer |

## Return value

An integer value that indicates the success of the function:

- 0 The real fields were not created in the message buffer.
- 1 The real fields were successfully created in the message buffer.

## MMSyncSendMessageNumStrings function

### Description

Use the `MMSyncSendMessageNumStrings` function to define the number and length of string fields in the send message buffer. The number of string fields determines the number of string values that can be sent in a message from this machine to the MMS Server. The length of the string fields determines the maximum number of characters that can be stored in each field. To write a value to a string field, use the `MMSyncSendMessageString` function. For more information, see “`MMSyncSendMessageString` function” on [page D.16](#).

---

**Tip** Instead of calling a separate function to define the number of fields for each data type, you can define the number of integer, real, and string fields in the send message buffer using one function. For more information, see “`MMSyncSendMessageNumDatas` function” on [page D.11](#).

---

### Parameters

The `MMSyncSendMessageNumStrings` function requires two parameters:

`MMSyncSendMessageNumStrings(<String_fields>, <String_length>)`

The parameters are defined in the following table:

| Parameter     | Type    | Description  |
|---------------|---------|--|
| String_fields | Integer | The number of string fields to create in the send message buffer                                   |
| String_length | Integer | The length of each string field (the number of characters that can be stored in each string field) |

### Return value

An integer value that indicates the success of the function:

- 0 The string fields were not created in the message buffer.
- 1 The string fields were successfully created in the message buffer.

## MMSyncSendMessagePrint function

### Description

Use the `MMSyncSendMessagePrint` function to print the content of the current send message buffer to a string expression. If the trace level of a model or the MMS Server is greater than or equal to one, this function is automatically called after the `MMSyncSendMessage` function, and the return value is written to a trace file. For more information, see “Using trace files” on [page 4.25](#).

### Parameters

The `MMSyncSendMessagePrint` function has no parameters.

### Return value

A tab-delimited string containing the following information:

- The current simulation time.
- The message type.
- The values of integer fields in the send message buffer.
- The values of real fields in the send message buffer.
- The values of string fields in the send message buffer.

## MMSyncSendMessageReal function

### Description

Use the `MMSyncSendMessageReal` function to write values to the real fields of the message contained in the send message buffer. Call the function once for each field to which you want to write a value. You can change the number of real fields in the send message buffer using the `MMSyncSendMessageNumReals` function (for more information, see “`MMSyncSendMessageNumReals` function” on [page D.13](#)).

### Parameters

The `MMSyncSendMessageReal` function requires two parameters:

`MMSyncSendMessageReal(<Index>, <Value>)`

The parameters are defined in the following table:

| Parameter | Type    | Description  |
|-----------|---------|--|
| Index     | Integer | The index of the real field to which you are writing a value |
| Value     | Real    | The value that is written to the real field                  |

### Return value

An integer value that indicates the success of the function:

- 0 The value was not written to the real field.
- 1 The value was successfully written to the real field.

---

**Note** If the return value of the function is zero, check the value of the first parameter to make sure you are specifying a valid real field in the message. Another reason the function may fail is if the model is not connected to the MMS Server.

---

## MMSyncSendMessageString function

### Description

Use the `MMSyncSendMessageString` function to write values to the string fields of the message contained in the send message buffer. Call the function once for each field to which you want to write a value. You can change the number of string fields in the send message buffer using the `MMSyncSendMessageNumStrings` function (for more information, see “`MMSyncSendMessageNumStrings` function” on [page D.14](#)).

## Parameters

The `MMSyncSendMessageString` function requires two parameters:

`MMSyncSendMessageString(<Index>, <Value>)`

The parameters are defined in the following table:

| Parameter | Type    | Description  |
|-----------|---------|--|
| Index     | Integer | The index of the string field to which you are writing a value |
| Value     | String  | The value that is written to the string field                  |

## Return value

An integer value that indicates the success of the function:

- 0 The value was not written to the string field.
- 1 The value was successfully written to the string field.

---

**Note** If the return value of the function is zero, check the value of the first parameter to make sure you are specifying a valid string field in the message. Another reason the function may fail is if the model is not connected to the MMS Server.

---

## MMSyncSendMessageType function

### Description

Use the `MMSyncSendMessageType` function to define the type of the message contained in the send message buffer. Using MMS message types, you can anticipate the structure of a message, making message interpretation easier. For more information, see “Defining the type of an MMS message” on [page 4.10](#).

### Parameters

The `MMSyncSendMessageType` function requires one parameter:

`MMSyncSendMessageType(<Type>)`

The parameter is defined in the following table:

| Parameter | Type    | Description  |
|-----------|---------|--|
| Type      | Integer | An integer defining the type of the message contained in the send message buffer (must be greater than or equal to 5000) |

### Return value

An integer value that indicates the success of the function:

- 0 The message type was not set.
- 1 The message type was set successfully.

---

## User-defined MMS functions

AutoMod provides one predeclared MMS function that you can define in a model. During a model run, the software calls this function automatically.

### model mmsyncmessage function

#### Description

Use the `model mmsyncmessage` function to read and interpret a message from the read message buffer. The function is called each time a message is received from the MMS Server and stored in the read message buffer. For more information, see “Reading an MMS message” on [page 4.13](#).

If models are exactly synchronized, the function is called in the receiving model at the same simulation time as when the message was sent by the originating model.

If models are periodically synchronized, the function is called at the time of the earliest synchronization update after the message is sent by the originating model. For more information, see “Synchronizing simulation clocks” on [page 4.4](#).

- ! **Important** Functions that are used to read from the read message buffer can only be called from within the `model mmsyncmessage` function.

#### Parameters

The `model mmsyncmessage` function has no parameters.



## **Return value**

You can return any integer value; the software ignores the return value.



# Glossary

**active model** A synchronized model that advances to the time of its next future event and completes the events on its current event list.

**blocking attempt/accept** The client's connection attempts or the server's acceptances repeat continuously until a connection is opened.

**blocking socket** A socket at which read attempts repeat continuously until a message is read.

**C structure message** A message composed of values of different data types stored under one name. C structure messages are not available in AutoMod code and must be sent from a user-defined C function.

**client** A program that connects to a server, usually through a network connection, and relies on the server to perform some operations. An AutoMod model can connect as a client to the MMS Server (which synchronizes models and distributes messages to models) or to an OPC server (which stores data about a system's current conditions and can update clients when data changes). It is also possible for a model that is communicating through sockets to connect as a client to another model that is acting as a server.

**communication event** An action or occurrence that the AutoMod model or the OPC server can detect. For example, communication events include all values that are read from and written to the server.

**exact synchronization** A method of synchronization in which only one model is active while the other models delay. When

models are exactly synchronized, the future event list of each model in the simulation is continually compared to ensure that the model with the earliest scheduled future event is the active model during simulation.

**field** A space in an MMS message that is allocated for a value of a particular data type. The default number of fields in a message are eight fields of each type (String, Integer, and Real).

**group** A client-specific collection of items. Each client application that connects to an OPC server can create its own groups for organizing the items on the server. Groups have properties that can be used to limit the amount of item updates that are sent from the server to the client.

**interpreting a message** The process of writing user-defined functions or procedures that define which actions should be performed based on the content of a message.

**item** A variable that is stored on an OPC server. Depending on the item's access rights, a client can read and write values to the item on the server.

**member** A variable in a structure written in C code.

**message** Any data that can be contained in a string variable or a C structure.

**MMS message** A message used to communicate between models and the Multi-Model Synchronization Server (MMS Server). MMS messages can consist of three different types of data: string, integer, and real. MMS messages that are sent by a

model to the MMS Server are broadcast by the server to each of the synchronized models participating in the simulation.

**MMS message type** An integer greater than or equal to 5000 that is used to identify the purpose and structure of incoming messages.

**MMS Server** A program that automatically synchronizes AutoMod models and distributes communications between the models during a simulation.

**OPC server** A program that facilitates the sharing of data between a system's industrial devices and various software applications, without the need for proprietary drivers. A model can connect as a client to an OPC server and emulate system devices by providing the same responses to the OPC server as the devices in a real system.

**OPCENUM** A component supplied by the OPC Foundation that allows client applications to obtain a list of available servers on a local or remote machine. The OPC Utility and the `OPCListItems` function both use this utility to get information about available OPC servers.

**packet** A bundle of data that varies in size depending on your network hardware, but is usually less than 1500 bytes.

**periodic synchronization** A method of synchronization in which models are synchronized at regular intervals during a simulation; the models run asynchronously between each synchronization update.

**port number** A number greater than 1000 that is used by TCP to match the client's connection attempt with the server's acceptance. The port number must be the same for both the client and server.

**read message buffer** A temporary storage area in RAM where the structure, type, and content of an incoming MMS message is stored.

**RSLink** A Rockwell communications server that communicates between plant devices and software applications.

**send message buffer** A temporary storage area in RAM where the structure, type, and content of an outgoing MMS message are defined.

**server** A program that manages network resources. The Model Communications module allows models that are communicating through sockets to act as a server for one or more client models. Models can also connect as clients to the MMS Server or an OPC server.

**service name** An alphabetic reference to a port number. For example, the service name `model1` can be used to represent port # 3554.

**socket** A communication point through which applications can send and receive data.

**socket pointer** A standard AutoMod type for variables and load attributes that allows you to point to the socket connection through which you want to transfer data.

**stream socket** A type of socket that can send and receive data. Stream sockets send data in sequenced, unduplicated packets.

**string message** A message composed of one or more alphanumeric characters. String messages are the only message format that can be used in standard AutoMod code. The content of a string message is all of the same type (type string).

**structure** A collection of related variables that are stored under one name.

**subscription** A method of reading item values only when they change on an OPC server. After a model subscribes to one or more items on the server, the server continuously scans the values of the subscribed items and updates the model when the values change.

**synchronization** The process of delaying models so that the simulation clock advances in all models at the same rate.

**timeout attempt/accept** The client's connection attempts or the server's acceptances repeat for a specified length of time; if a connection is not opened within the defined time period, the client or server stops trying to open a connection.

**timeout socket** A socket at which read attempts repeat for a specified length of time; if a sent message does not arrive within the defined time period, the function stops trying to read a message and must be called again.

**trace file** A tab-delimited text file that contains a list of communication events that occurred during a model run. A model can create a trace file when communicating with either an OPC server or the MMS Server. You can duplicate the results of a traced run by running a model with the trace file (this allows you to run a simulation without needing a live connection to a server).

**Transmission Control Protocol (TCP)** A reliable byte-stream transfer service between sockets. TCP protects against data loss and data corruption. TCP also provides flow-control that continually adjusts the rate at which data is sent in an attempt to achieve maximum data throughput while avoiding network congestion and packet loss.



# Index

## Symbols

~ (comments) 4.19

## A

accepting connections 3.10  
     continuously 3.11  
 AcceptSocketPort function 3.10, C.1  
 AcceptSocketService function 3.10, C.2  
 access rights, item 2.6, 2.26, 2.30  
 active model 4.4  
 adding  
     groups 2.16  
     items 2.17  
 architecture, process control 2.1  
 asynchronous simulation clocks 4.3  
 attempting connections 3.8  
 AutoStat 2.55

## B

blocking  
     connections 3.3  
     reading messages 3.20  
     sockets 3.20  
 Boolean items  
     reading 2.31  
     writing 2.25  
 buffers  
     read message buffer 4.13  
     printing content of 4.15  
     send message buffer 4.7  
     printing content of 4.11  
 byte order, sorting 3.14

## C

C structure messages 3.13  
     interpreting 3.25

    reading 3.21  
     sending 3.17  
 callbacks 2.37, 2.47  
 CBack (statistic) 2.47  
 client 3.2  
     attempting a connection 3.8  
 CloseSocket function 3.13, C.3  
 closing connections  
     OPC 2.25  
     sockets 3.13, 3.17–3.18  
 codes  
     quality A.1  
 command line options  
     MMS Server 4.23  
 configuration file, creating 2.9  
 connections  
     accepting 3.10  
     attempting 3.8  
     blocking 3.3  
     client 3.8  
     closing  
         OPC 2.25  
         sockets 3.13, 3.17–3.18  
     multiple 3.7  
     opening  
         OPC 2.9–2.22  
         sockets 3.2  
     server 3.10  
     timeout 3.3  
     verifying  
         OPC 2.22  
         sockets 3.11  
 ConnectSocketPort function 3.8, C.3  
 ConnectSocketService function 3.8  
 control systems  
     PLCs 2.1  
     synchronizing a model with 2.41  
 converting numeric values 3.14

current event list 4.3

## D

Data (statistic) 2.47

deadband 2.7

delays, real time 2.44

displaying

connection information 2.22

item values 2.46

DNS server 3.3

Domain Name System server 3.3

## E

emulation 1.1, 2.1

running a model 2.40

versus simulation 2.8

event lists

current 4.3

future 4.3

events 4.2

synchronizing 4.4

exact synchronization

with the MMS Server 4.4, 4.23

without the MMS Server 3.26

## F

FEvFirstTime function 3.28, C.5

fields

number of

defining 4.8

reading 4.14

values

defining 4.9

reading 4.15

files

hosts 3.3

model data 4.19

trace

MMS 4.25, 4.28

OPC 2.48, 2.54

float items

reading 2.31

writing 2.25

functions

AcceptSocketPort 3.10, C.1

AcceptSocketService 3.10, C.2

CloseSocket 3.13, C.3

ConnectSocketPort 3.8, C.3

ConnectSocketService 3.8

FEvFirstTime 3.28, C.5

GetSocketHost C.5

GetSocketName 3.12, C.5

GetSocketNum C.6

htons C.7

MachineName 3.9, C.6

MilliSleep 2.44, B.1

MMSyncReadMessageInteger 4.15, D.1

MMSyncReadMessageNumIntegers 4.14,

D.2

MMSyncReadMessageNumReals 4.14, D.3

MMSyncReadMessageNumStrings 4.14,

D.3

MMSyncReadMessagePrint 4.15, D.4

MMSyncReadMessageReal 4.15, D.4

MMSyncReadMessageString 4.15, D.5

MMSyncReadMessageStringLength 4.14,

D.6

MMSyncReadMessageTime 4.15, D.7

MMSyncReadMessageType 4.15, D.7

MMSyncSendMessage 4.12, D.8

MMSyncSendMessageClear 4.12, D.8

MMSyncSendMessageInteger 4.10, D.9

MMSyncSendMessageModel 4.11, D.10

MMSyncSendMessageNumDatas 4.8, D.11

MMSyncSendMessageNumIntegers 4.8,

D.12

MMSyncSendMessageNumReals 4.8, D.13

MMSyncSendMessageNumStrings 4.8,

D.14

MMSyncSendMessagePrint 4.11, D.15

MMSyncSendMessageReal 4.10, D.16

MMSyncSendMessageString 4.10, D.16

MMSyncSendMessageType 4.10, D.17

model mmsyncmessage 4.13, D.18

model nextclock 3.11, 3.27, C.19

NetworkConvertDouble 3.15, C.8

NetworkConvertFloat 3.15, C.8

NetworkConvertInteger 3.15, C.7

NetworkConvertShort 3.15, C.7

OPCAddGroup 2.16, B.2

OPCAddItem 2.17, B.3

opccallback 2.33, 2.37, B.23

OPCConnectServer 2.15, B.5

OPCDisconnect 2.25, B.6

OPCFlushWrite 2.28, B.7

OPCInit 2.11, B.8

OPCListConnections 2.23, B.9

OPCListItems 2.13, B.10

OPCReadInteger 2.31, B.11



OPCReadReal 2.31, B.11  
 OPCReadString 2.31, B.12  
 OPCRefreshSubscription 2.35, B.13  
 OPCSubscribe 2.33, B.14  
 OPCWriteInteger 2.28, B.16  
 OPCWriteIntegerWait 2.26, B.17  
 OPCWriteReal 2.28, B.18  
 OPCWriteRealWait 2.26, B.20  
 OPCWriteString 2.28, B.21  
 OPCWriteStringWait 2.26, B.22  
 ReadSocket 3.19, 3.21, C.9  
 ReadSocketString 3.19–3.20, C.9  
 SendSocket 3.16, C.10  
 SendSocketString 3.16, C.11  
 SetAcceptBlocking 3.4, C.11  
 SetAcceptTimeOut 3.4, C.12  
 SetConnectBlocking 3.4, C.13  
 SetConnectTimeOut 3.4, C.14  
 SetNumAcceptsPort 3.8, C.14  
 SetNumAcceptsService 3.8, C.15–C.16  
 SetSocketMessages 3.10–3.11, C.16  
 SetSocketNonBlocking 3.20, C.16  
 SetSocketTimeOut 3.20, C.17  
 SetSyncRate 2.41  
 ValidSocket 3.12, C.18

future events  
     future event list 4.3

## G

GetSocketHost function C.5  
 GetSocketName function 3.12, C.5  
 GetSocketNum function C.6  
 Group (statistic) 2.47

groups 2.3  
     adding 2.16  
     displaying 2.22  
     printing 2.23  
     properties 2.6

## H

hierarchy, OPC servers 2.3  
 hosts file 3.3  
 hton function C.7

## I

integer fields  
     number of  
         defining 4.8

        reading 4.14  
     values  
         defining 4.10  
         reading 4.15

integer items  
     reading 2.31  
     writing 2.25

interpreting messages  
     C structure 3.25  
     MMS 4.17  
     string 3.24

IP addresses 3.3

Item (statistic) 2.47

items 2.3  
     access rights 2.26, 2.30  
     adding 2.17  
     defining a reference to B.26  
     displaying 2.22  
     displaying values 2.46  
     printing 2.23  
     printing to the Messages window 2.13  
     properties 2.5  
     subscribing 2.33  
     types 2.5  
     update rate 2.7  
     verifying values 2.28  
     writing values 2.25

## L

level, trace  
     MMS Server 4.24  
     synchronized models 4.21

linked variables 2.11, 2.19

list  
     current event list 4.3  
     future event list 4.3

## M

MachineName function 3.9, C.6  
 machines, defining a reference to B.26  
 members 3.13  
 messages 3.1  
     C structure 3.13  
     formats 3.13  
     interpreting 3.23, 4.17  
     MMS 4.7  
     reading 3.19, 4.13  
     sending 3.16, 4.7

- string 3.13
  - turning off warnings 3.10– 3.11
  - MilliSleep function 2.44, B.1
  - MMS
    - functions
      - MMSyncReadMessageInteger 4.15, D.1
      - MMSyncReadMessageNumIntegers 4.14, D.2
      - MMSyncReadMessageNumReals 4.14, D.3
      - MMSyncReadMessageNumStrings 4.14, D.3
      - MMSyncReadMessagePrint 4.15, D.4
      - MMSyncReadMessageReal 4.15, D.4
      - MMSyncReadMessageString 4.15, D.5
      - MMSyncReadMessageStringLength 4.14, D.6
      - MMSyncReadMessageTime 4.15, D.7
      - MMSyncReadMessageType 4.15, D.7
      - MMSyncSendMessage 4.12, D.8
      - MMSyncSendMessageClear 4.12, D.8
      - MMSyncSendMessageInteger 4.10, D.9
      - MMSyncSendMessageModel 4.11, D.10
      - MMSyncSendMessageNumDatas 4.8, D.11
      - MMSyncSendMessageNumIntegers 4.8, D.12
      - MMSyncSendMessageNumReals 4.8, D.13
      - MMSyncSendMessageNumStrings 4.8, D.14
      - MMSyncSendMessagePrint 4.11, D.15
      - MMSyncSendMessageReal 4.10, D.16
      - MMSyncSendMessageString 4.10, D.16
      - MMSyncSendMessageType 4.10, D.17
      - model mmsyncmessage 4.13, D.18
    - messages 4.7
      - interpreting 4.17
      - reading 4.13
      - recipient 4.11
      - sending 4.7
      - structure 4.8, 4.14
      - type 4.10, 4.15
    - MMSserver.trace 4.23
    - Server 1.4
      - ports 4.23
      - service name 4.23
      - starting 4.22
      - synchronizing without 3.26
  - models
    - active 4.4
    - Model Communications module 1.2
    - Model Communications Plus module 1.4
    - model data file 4.19
    - model mmsyncmessage function 4.13, D.18
    - model nextclock function 3.11, 3.27, C.19
    - models.mms 4.24
    - modular 1.2
    - running with a trace file 2.54, 4.28
    - sending and reading messages. *See* messages
    - synchronizing 4.1
    - synchronizing without the MMS Server 3.26
  - modular simulations 1.2
  - modules
    - Model Communications 1.2
    - Model Communications Plus 1.4
  - Multi-model synchronization *See* MMS
  - multiple connections 3.7
- ## N
- names
    - server model machine C.6
  - NetworkConvertDouble function 3.15, C.8
  - NetworkConvertFloat function 3.15, C.8
  - NetworkConvertInteger function 3.15, C.7
  - NetworkConvertShort function 3.15, C.7
  - numeric values, converting 3.14
- ## O
- OLE for Process Control *See* OPC
  - OPC 1.3
    - functions
      - MilliSleep 2.44, B.1
      - OPCAddGroup 2.16, B.2
      - OPCAddItem 2.17, B.3
      - opccallback 2.33, 2.37, B.23
      - OPCConnectServer 2.15, B.5
      - OPCDisconnect 2.25, B.6
      - OPCFlushWrite 2.28, B.7
      - OPCInit 2.11, B.8
      - OPCListConnections 2.23, B.9
      - OPCListItems 2.13, B.10
      - OPCReadInteger 2.31, B.11
      - OPCReadReal 2.31, B.11
      - OPCReadString 2.31, B.12
      - OPCRefreshSubscription 2.35, B.13
      - OPCSubscribe 2.33, B.14
      - OPCWritelInteger 2.28, B.16
      - OPCWritelIntegerWait 2.26, B.17
      - OPCWriteReal 2.28, B.18

- OPCWriteRealWait 2.26, B.20
- OPCWriteString 2.28, B.21
- OPCWriteStringWait 2.26, B.22
- SetSyncRate 2.41
- OPC Utility 2.9
- OPCQuality 2.30
- OPCTimestamp 2.30
- quality codes A.1
- servers 2.3
  - adding groups 2.16
  - adding items 2.17
  - deadband 2.7
  - displaying connections 2.22
  - printing connections 2.23
  - RSLinX 2.21
  - scan rate 2.6
  - update rate 2.7
- OPCFlushWrite function 2.28
- OPCType (trace file) 2.50–2.51
- opening connections
  - OPC 2.9–2.22
  - sockets 3.2, 3.7
- options, command line
  - MMS Server 4.23
  - synchronized models 4.20

## P

- packets 3.1
- paths
  - to model executables 4.20
  - to the model data file 4.24
- periodic synchronization 4.6, 4.23
- PLCs 2.1, 2.41
- pointers, sockets 3.11
- ports
  - MMS Server 4.23
  - multiple connections 3.8
  - numbers 3.6
- printing
  - connection information 2.23
  - content of read message buffer 4.15
  - content of send message buffer 4.11
  - items 2.13
  - socket information 3.11
- process control architecture 2.1
- Process Server
  - starting 4.18

- programmable logic controllers 2.1
- properties
  - groups 2.6
  - items 2.5

## Q

- Qlty (statistic) 2.47–2.48
- quality of OPC item values 2.30
- quality, item 2.5, 2.38

## R

- rate
  - item update 2.7
  - server scan 2.6
- reading item values 2.30–2.40
- reading messages
  - blocking 3.20
  - C structure 3.21
  - MMS 4.13
  - read message buffer 4.13
    - printing content of 4.15
  - string 3.21
  - timeout 3.20
- Reads (statistic) 2.47
- ReadSocket function 3.19, 3.21, C.9
- ReadSocketString function 3.19–3.20, C.9
- real fields
  - number of
    - defining 4.8
    - reading 4.14
  - values
    - defining 4.10
    - reading 4.15
- real items
  - reading 2.31
  - writing 2.25
- real time delays 2.44
- recipient of MMS messages 4.11
- references
  - to a machine B.26
  - to an item B.26
- refreshing subscriptions 2.35
- remote machines
  - setting up for synchronization 4.18
- RSLinX 2.21

running an emulation model 2.40

runs made with a trace file

MMS 4.28

OPC 2.54

## S

scenarios, testing 2.55

sending messages

C structure 3.17

MMS 4.7

to a specific model 4.11

send message buffer 4.7

printing content of 4.11

string 3.16

SendSocket function 3.16, C.10

SendSocketString function 3.16, C.11

Server (statistic) 2.46

servers

MMS 1.4

starting 4.22

models 3.2

accepting a connection 3.10

machine name C.6

OPC 2.3

adding groups 2.16

adding items 2.17

deadband 2.7

displaying connections 2.22–2.23

RSLink 2.21

scan rate 2.6

update rate 2.7

Process

starting 4.18

service name

accepting a connection 3.11

attempting a connection 3.9

defining 3.6

MMS Server 4.23

set link 2.11, 2.19

SetAcceptBlocking function 3.4, C.11

SetAcceptTimeOut function 3.4, C.12

SetConnectBlocking function 3.4, C.13

SetConnectTimeOut function 3.4, C.14

SetNumAcceptsPort function 3.8, C.14

SetNumAcceptsService function 3.8, C.15–C.16

SetSocketMessages function 3.10–3.11, C.16

SetSocketNonBlocking function 3.20, C.16

SetSocketTimeOut function 3.20, C.17

SetSyncRate function 2.41

simulation clocks

synchronizing 4.4

simulations, modular 1.2

sockets 1.3

blocking 3.20

pointers 3.11

setting pointers to null 3.17

stream 3.1

timeout 3.20

sorting byte order 3.14

starting

MMS Server 4.22

synchronized simulations 4.18

statistics, update 2.46

stream sockets 3.1

string fields

number of

defining 4.8

reading 4.14

values

defining 4.10

reading 4.15

string items

reading 2.31

writing 2.25

string messages 3.13

interpreting 3.24

reading 3.21

sending 3.16

structure

C *See* C structure messages

MMS messages

defining 4.8

reading 4.14

subscribe link 2.11, 2.19

subscriptions

callbacks 2.37

refreshing 2.35

responding to changed values 2.37

subscribing to items 2.33

updates 2.7

Synchronization 1.3

synchronization 4.1

- exact 3.26, 4.4, 4.23
- periodic 4.6, 4.23
- starting a simulation 4.18
- with control systems 2.41
- without the MMS Server 3.26

synchronous write 2.25

## T

TCP 3.2

testing scenarios 2.55

tilde comments 4.19

timeout

- connections 3.4
- reading messages 3.20
- sockets 3.20

Timestamp (statistic) 2.47–2.48

timestamp, item 2.6, 2.30, 2.38

tolerance, real value 2.26

trace

- files
  - MMS 4.25, 4.28
  - OPC 2.48, 2.54
- level
  - MMS Server 4.24
  - synchronized models 4.21

transmission control protocol. *See* TCP

types

- MMS messages
  - defining 4.10
  - reading 4.15
- numeric conversions 3.15
- OPC items 2.5

## U

update link 2.11, 2.19

update rate 2.7

update statistics 2.46

Utility, OPC 2.9

## V

ValidSocket function 3.12, C.18

values, items 2.5

- reading 2.30
- refreshing 2.35
- writing 2.25

variables, linked 2.11, 2.19

verifying connections

- OPC 2.22
- sockets 3.11

verifying written item values 2.28

## W

warning messages, turning off 3.10–3.11

Writes (statistic) 2.47

writing item values 2.25

