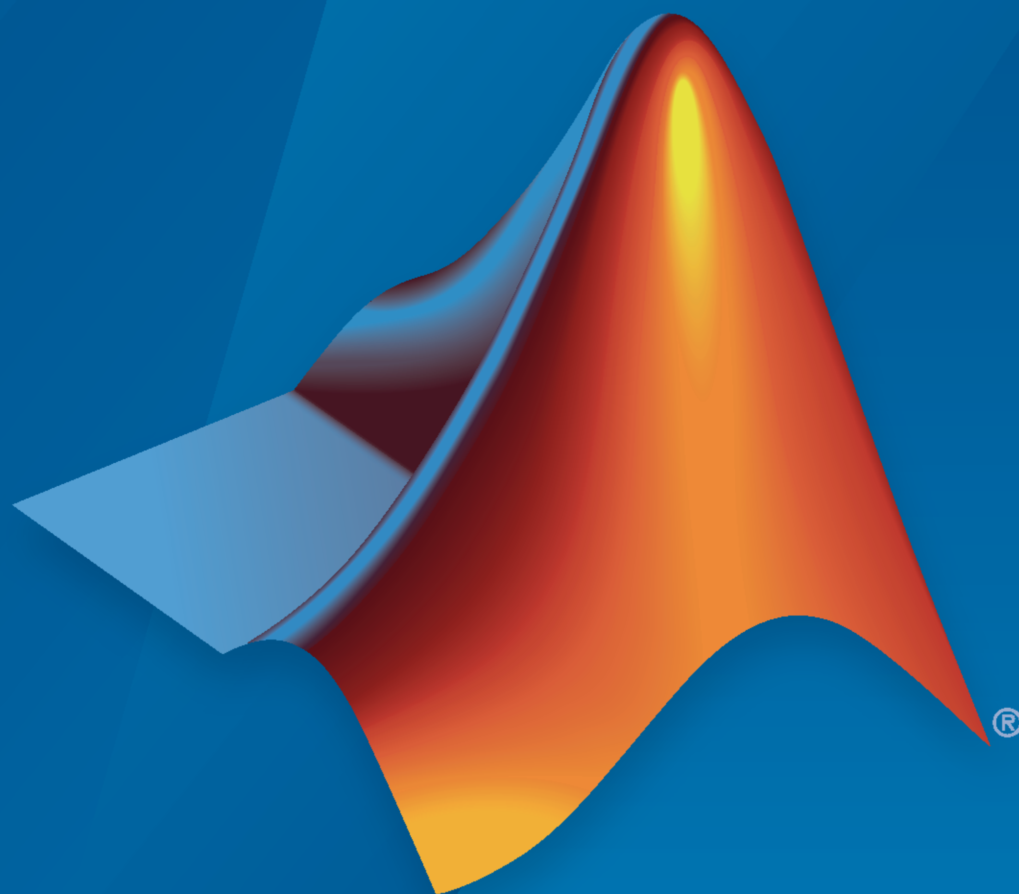
Instrument Control Toolbox™

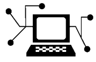
User's Guide



MATLAB® & SIMULINK®

R2021b

How to Contact MathWorks



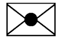
Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000



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

Instrument Control Toolbox™ User's Guide

© COPYRIGHT 2005–2021 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

November 2000	First printing	New for Version 1.0 (Release 12)
June 2001	Second printing	Revised for Version 1.1 (Release 12.1)
July 2002	Online only	Revised for Version 1.2 (Release 13)
August 2002	Third printing	Revised for Version 1.2
June 2004	Online only	Revised for Version 2.0 (Release 14)
October 2004	Fourth printing	Revised for Version 2.1 (Release 14SP1)
March 2005	Online only	Revised for Version 2.2 (Release 14SP2)
June 2005	Fifth printing	Minor revision for Version 2.2
September 2005	Online only	Revised for Version 2.3 (Release 14SP3)
March 2006	Online only	Revised for Version 2.4 (Release 2006a)
September 2006	Online only	Revised for Version 2.4.1 (Release 2006b)
March 2007	Online only	Revised for Version 2.4.2 (Release 2007a)
September 2007	Sixth printing	Revised for Version 2.5 (Release 2007b)
March 2008	Online only	Revised for Version 2.6 (Release 2008a)
October 2008	Online only	Revised for Version 2.7 (Release 2008b)
March 2009	Online only	Revised for Version 2.8 (Release 2009a)
September 2009	Online only	Revised for Version 2.9 (Release 2009b)
March 2010	Online only	Revised for Version 2.10 (Release 2010a)
September 2010	Online only	Revised for Version 2.11 (Release 2010b)
April 2011	Online only	Revised for Version 2.12 (Release 2011a)
September 2011	Online only	Revised for Version 3.0 (Release 2011b)
March 2012	Online only	Revised for Version 3.1 (Release 2012a)
September 2012	Online only	Revised for Version 3.2 (Release 2012b)
March 2013	Online only	Revised for Version 3.3 (Release 2013a)
September 2013	Online only	Revised for Version 3.4 (Release 2013b)
March 2014	Online only	Revised for Version 3.5 (Release 2014a)
October 2014	Online only	Revised for Version 3.6 (Release 2014b)
March 2015	Online only	Revised for Version 3.7 (Release 2015a)
September 2015	Online only	Revised for Version 3.8 (Release 2015b)
March 2016	Online only	Revised for Version 3.9 (Release 2016a)
September 2016	Online only	Revised for Version 3.10 (Release 2016b)
March 2017	Online only	Revised for Version 3.11 (Release 2017a)
September 2017	Online only	Revised for Version 3.12 (Release 2017b)
March 2018	Online only	Revised for Version 3.13 (Release 2018a)
September 2018	Online only	Revised for Version 3.14 (Release 2018b)
March 2019	Online only	Revised for Version 4.0 (Release 2019a)
September 2019	Online only	Revised for Version 4.1 (Release 2019b)
March 2020	Online only	Revised for Version 4.2 (Release 2020a)
September 2020	Online only	Revised for Version 4.3 (Release 2020b)
March 2021	Online only	Revised for Version 4.4 (Release 2021a)
September 2021	Online only	Revised for Version 4.5 (Release 2021b)

Contents

1 Getting Started

Instrument Control Toolbox Product Description 1-2

Instrument Control Toolbox Overview 1-3
Getting to Know the Instrument Control Toolbox Software 1-3
Exploring the Instrument Control Toolbox Software 1-3
Learning About the Instrument Control Toolbox Software 1-4
Using the Documentation Examples 1-4

About Instrument Control 1-5
Passing Information Between the MATLAB Workspace and Your Instrument 1-5
MATLAB Functions 1-6
Interface Driver Adaptor 1-6

Installation Information 1-8
Installation Requirements 1-8
Toolbox Installation 1-8
Hardware and Driver Installation 1-8

Supported Hardware 1-9

Examining Your Hardware Resources 1-10
instrhwinfo Function 1-10
Test & Measurement Tool 1-13
Viewing the IVI Configuration Store 1-14

Communicating with Your Instrument 1-16
Instrument Control Session Examples 1-16
Communicating with a GPIB Instrument 1-16
Communicating with a GPIB-VXI Instrument 1-17
Communicating with a Serial Port Instrument 1-17
Communicating with a GPIB Instrument Using a Device Object 1-18

General Preferences for Instrument Control 1-20
Accessing General Preferences 1-20
MATLAB Instrument Driver Editor 1-21
MATLAB Instrument Driver Testing Tool 1-21
Device Objects 1-22
IVI Configuration Store 1-22
IVI Instruments 1-23

Interface and Property Help 1-24
instrhelp Function 1-24
propinfo Function 1-24

instrsupport Function 1-25
Overview Help 1-25
Documentation Examples 1-26
Online Support 1-26

2 Instrument Control Session

Creating Instrument Objects **2-2**
Overview 2-2
Interface Objects 2-2
Device Objects 2-2

Connecting to the Instrument **2-3**

Configuring and Returning Properties **2-4**
Configuring Property Names and Property Values 2-4
Returning Property Names and Property Values 2-4
Using Tab Completion for Functions 2-4
Property Inspector 2-6

Communicating with Your Instrument **2-8**
Interface Objects and Instrument Commands 2-8
Device Objects and Instrument Drivers 2-8

Disconnecting and Cleaning Up **2-9**
Disconnecting an Instrument Object 2-9
Cleaning Up the MATLAB Workspace 2-9

Summary **2-10**
Advantages of Using Device Objects 2-10
When to Use Interface Objects 2-10

Instrument Control Toolbox Properties **2-11**

3 Using Interface Objects

Create Interface Object **3-2**
Connect to Instrument 3-2
Configure Properties 3-2

Configure and Return Properties **3-3**
Set Property Values During Object Creation and View Properties 3-3
Set Property Values 3-4

Write and Read Data **3-5**
Before Performing Write or Read Operations 3-5
Writing Data 3-5

Reading Data . . . 3-7

Use SCPI Commands . . . 3-9

4 Controlling Instruments Using GPIB

GPIB Overview . . . **4-2**
What Is GPIB? . . . **4-2**
Important GPIB Features . . . **4-2**
GPIB Lines . . . **4-3**
Status and Event Reporting . . . **4-6**

Write and Read GPIB Data . . . **4-10**
Rules for Completing Write and Read Operations . . . **4-10**
Writing and Reading Text Data . . . **4-10**
Writing and Reading Binary Data . . . **4-12**
Parse Input String Data . . . **4-14**

Transition Your Code to VISA-GPIB Interface . . . **4-16**
Discover GPIB Instruments . . . **4-16**
Connect to GPIB Instrument . . . **4-17**
Write and Read Binary or String Data . . . **4-17**
Read Terminated String . . . **4-18**
Read and Parse String Data . . . **4-18**
Write and Read Back Data . . . **4-19**
Write and Read Binblock Data . . . **4-19**
Flush Data from Memory . . . **4-20**
Set Terminator . . . **4-20**
Set Up Callback Function . . . **4-20**

5 Controlling Instruments Using VISA

Get Started with VISA . . . **5-2**
What Is VISA? . . . **5-2**
Supported Platforms and Minimum Driver Requirements . . . **5-2**
Interfaces Used with VISA . . . **5-2**
Connect to and Configure VISA Resource . . . **5-3**

Get Started with TCP/IP Interface for VXI-11 and HiSLIP . . . **5-5**
Create VISA-TCP/IP Object . . . **5-5**

Get Started with TCP/IP Socket Interface . . . **5-7**
Create VISA-Socket Object . . . **5-7**

Get Started with USB Interface . . . **5-9**
Create VISA-USB Object . . . **5-9**

Get Started with GPIB Interface **5-11**
Install Required Drivers **5-11**
Create VISA-GPIB Object **5-11**

Get Started with Serial Port Interface **5-13**
Create VISA-Serial Object **5-13**
Configure Communication Settings **5-14**

Get Started with VXI and PXI Interfaces **5-15**
Create VISA-VXI Object **5-15**
Create VISA-PXI Object **5-16**

Write and Read ASCII Data Using VISA **5-19**
Connect to Instrument **5-19**
Write ASCII Data **5-19**
Read ASCII Data **5-20**
Clean Up **5-21**

Write and Read Binary Data Using VISA **5-22**
Connect to Instrument **5-22**
Write Binary Data **5-22**
Read Binary Data **5-23**
Clean Up **5-23**

Use Callbacks for VISA Communication **5-24**
Use Events and Callbacks **5-24**
Event Types and Callback Properties **5-24**
Use Events and Callbacks to Display Event Information **5-25**

Send Trigger Message **5-27**
Use visatrigger Function **5-27**
Execute Trigger **5-27**

Execute Serial Polls **5-29**
Use visastatus Function **5-29**
Execute Serial Poll **5-29**

Transition Your Code to visadev Interface **5-32**
Discover VISA Devices **5-32**
Connect to VISA Device **5-33**
Write and Read Binary or String Data **5-33**
Read Terminated String **5-34**
Read and Parse String Data **5-34**
Write and Read Back Data **5-35**
Write and Read Binblock Data **5-35**
Flush Data from Memory **5-36**
Set Terminator **5-36**
Set Up Callback Function **5-36**

6 Controlling Instruments Using the Serial Port

Serial Port Overview **6-2**
What Is Serial Communication? **6-2**
Serial Port Interface Standard **6-2**
Supported Platforms **6-3**
Connecting Two Devices with a Serial Cable **6-3**
Serial Port Signals and Pin Assignments **6-4**
Serial Data Format **6-6**
Find Serial Port Information for Your Platform **6-9**

Create Serial Port Object **6-13**
Create a Serial Port Object **6-13**
Serial Port Object Display **6-14**

Configure Serial Port Communication Settings **6-15**

Write and Read Serial Port Data **6-17**
Rules for Completing Write and Read Operations **6-17**
Writing and Reading Text Data **6-17**
Writing and Reading Binary Data **6-19**

Use Callbacks for Serial Port Communication **6-21**
Callback Properties **6-21**
Using Callbacks **6-21**

Use Serial Port Control Pins **6-22**
Control Pins **6-22**
Signaling the Presence of Connected Devices **6-22**
Controlling the Flow of Data: Handshaking **6-24**

Transition Your Code to serialport Interface **6-26**
Discover Serial Port Devices **6-26**
Connect to Serial Port Device **6-26**
Read and Write **6-27**
Send a Command **6-27**
Read a Terminated String **6-28**
Write Data with the Binary Block Protocol **6-29**
Read Data with the Binary Block Protocol **6-29**
Flush Data from Memory **6-29**
Set Terminator **6-30**
Set Up a Callback Function **6-30**
Read Serial Pin Status **6-31**
Set Serial DTR and RTS Pin States **6-31**

7 Controlling Instruments Using TCP/IP

TCP/IP Communication Overview **7-2**

Create TCP/IP Client and Configure Settings **7-3**
Create Object Using Host Name **7-3**
Create Object Using IP Address **7-3**
Set Timeout Property **7-4**
Set Connect Timeout Property **7-4**
View TCP/IP Object Properties **7-5**

Write and Read Data over TCP/IP Interface **7-7**
Write Data **7-7**
Read Data **7-7**
Acquire Data from Weather Station Server **7-8**
Read Page from Website **7-8**

Use Callbacks for TCP/IP Communication **7-10**

Configure Connection in TCP/IP Explorer **7-11**

Events and Callbacks **7-12**
Event Types and Callback Properties **7-12**
Responding To Event Information **7-12**
Using Events and Callbacks **7-14**

Rules for Completing Read and Write Operations over TCP/IP and UDP **7-15**
Completing Write Operations **7-15**
Completing Read Operations **7-15**

Basic Workflow to Read and Write Data over TCP/IP **7-17**

Read and Write ASCII Data over TCP/IP **7-19**
Functions and Properties **7-19**
Configuring and Connecting to the Server **7-20**
Writing ASCII Data **7-20**
ASCII Write Properties **7-21**
Reading ASCII Data **7-21**
ASCII Read Properties **7-22**
Cleanup **7-22**

Read and Write Binary Data over TCP/IP **7-23**
Functions and Properties **7-23**
Configuring and Connecting to the Server **7-24**
Writing Binary Data **7-25**
Binary Write Properties **7-25**
Configuring InputBufferSize **7-25**
Reading Binary Data **7-26**
Cleanup **7-26**

Asynchronous Read and Write Operations over TCP/IP **7-27**
Functions and Properties **7-27**
Synchronous Versus Asynchronous Operations **7-28**
Configuring and Connecting to the Server **7-28**
Reading Data Asynchronously **7-28**
Reading Data Asynchronously – Continuous ReadAsyncMode **7-29**
Reading Data Asynchronously – Manual ReadAsyncMode **7-29**
Defining an Asynchronous Read Callback **7-30**

Using Callbacks During an Asynchronous Read 7-30
Writing Data Asynchronously 7-31
Cleanup 7-31

TCP/IP and UDP Comparison 7-32
Supported Platforms 7-32
Interface Comparison 7-32

Communicate Using TCP/IP Server Sockets 7-34
About Server Sockets 7-34
Communicate Between Two Instances of MATLAB 7-34

Transition Your Code to tcpclient Interface 7-36
Create a TCP/IP Client 7-36
Write and Read 7-37
Read Terminated String 7-37
Read and Parse String Data 7-38
Write and Read Back Data 7-39
Write and Read Data with the Binary Block Protocol 7-39
Flush Data from Memory 7-40
Set Terminator 7-40
Set Up Callback Function 7-40

Transition Your Code to tcpserver Interface 7-42
Create a TCP/IP Server 7-42
Write and Read 7-43
Read Terminated String 7-43
Read and Parse String Data 7-44
Write and Read Data with the Binary Block Protocol 7-45
Flush Data from Memory 7-45
Set Terminator 7-46
Set Up Callback Function 7-46

8 Controlling Instruments Using UDP

Create a UDP Object and View Properties 8-2
Create a Byte-Type UDP Object and View Properties 8-2
Create a Datagram-Type UDP Object and View Properties 8-3
Enable Port Sharing over UDP 8-4

UDP Communication Between Two Hosts 8-5

Write and Read ASCII Data Over UDP 8-7
Configure and Connect to the Server 8-7
Write ASCII Data 8-7
Read ASCII Data 8-7
Clean Up 8-8

Write and Read Binary Data Over UDP 8-9
Write and Read Binary Data with Byte-Type UDP Port 8-9
Write and Read Binary Data with Datagram-Type UDP Port 8-10

Use Callbacks for UDP Communication **8-13**
Callback Properties **8-13**
Use Byte-Mode Callback with Byte-Type UDP Object **8-13**
Use Terminator-Mode Callback with Byte-Type UDP Object **8-14**
Use Datagram-Mode Callback with Datagram-Type UDP Object **8-16**

Transition Your Code to udpport Interface **8-18**
Connect to UDP Socket **8-18**
Read and Write **8-18**
Read Terminated String **8-20**
Read and Parse String Data **8-21**
Write and Read Back Data **8-21**
Flush Data from Memory **8-22**
Set Terminator **8-22**
Set Up Callback Function **8-22**

Basic Workflow to Read and Write Data over UDP **8-24**

Asynchronous Read and Write Operations over UDP **8-26**
Functions and Properties **8-26**
Synchronous Versus Asynchronous Operations **8-26**
Configuring and Connecting to the Server **8-27**
Reading Data Asynchronously **8-27**
Reading Data Asynchronously Using Continuous ReadAsyncMode **8-28**
Reading Data Asynchronously Using Manual ReadAsyncMode **8-28**
Defining an Asynchronous Read Callback **8-29**
Using Callbacks During an Asynchronous Read **8-29**
Writing Data Asynchronously **8-30**
Cleanup **8-30**

9 Controlling Instruments Using I2C

I2C Interface Overview **9-2**
I2C Communication **9-2**
Supported Platforms for I2C **9-2**

Configuring I2C Communication **9-3**
Configuring Total Phase Aardvark **9-3**
Configuring NI USB-845x **9-4**

Transmitting Data Over the I2C Interface **9-6**
Aardvark Example **9-6**
NI USB-845x Example **9-8**

I2C Interface Usage Requirements and Guidelines **9-10**
Aardvark-specific Requirements **9-10**
NI USB-845x-specific Requirements **9-10**

10 Controlling Instruments Using SPI

SPI Interface Overview **10-2**
SPI Communication **10-2**
Supported Platforms for SPI **10-2**

Configuring SPI Communication **10-3**

Transmitting Data Over the SPI Interface **10-7**

Using Properties on the SPI Object **10-13**

SPI Interface Usage Requirements and Guidelines **10-16**

11 Controlling Devices Using MODBUS

MODBUS Interface Supported Features **11-2**
MODBUS Capabilities **11-2**
Supported Platforms for MODBUS **11-2**

Create a MODBUS Connection **11-3**

Configure Properties for MODBUS Communication **11-5**

Read Data from a MODBUS Server **11-8**
Types of Data You Can Read Over MODBUS **11-8**
Reading Coils Over MODBUS **11-8**
Reading Inputs Over MODBUS **11-9**
Reading Input Registers Over MODBUS **11-9**
Reading Holding Registers Over MODBUS **11-10**
Specifying Server ID and Precision **11-10**
Reading Mixed Data Types **11-11**

Read Temperature from a Remote Temperature Sensor **11-13**

Write Data to a MODBUS Server **11-14**
Types of Data You Can Write to Over MODBUS **11-14**
Writing Coils Over MODBUS **11-14**
Writing Holding Registers Over MODBUS **11-14**

Write and Read Multiple Holding Registers **11-16**

Modify the Contents of a Holding Register Using a Mask Write **11-18**

Use the Modbus Explorer App **11-19**

Configure a Connection in the Modbus Explorer **11-20**
Communicate Over TCP/IP **11-20**
Communicate Over Serial RTU **11-21**

Read Coils, Inputs, and Registers in the Modbus Explorer **11-23**
Edit the Read Registers Table **11-24**
Import or Export Read Data **11-25**

Write to Coils and Holding Registers in the Modbus Explorer **11-26**

Control a PLC Using the Modbus Explorer **11-28**

Generate a Script from Your Modbus Explorer Session **11-33**

12 Using Device Objects

Device Objects **12-2**
Overview **12-2**
What Are Device Objects? **12-2**
Device Objects for MATLAB Instrument Drivers **12-3**

Creating and Connecting Device Objects **12-5**
Device Objects for MATLAB Interface Drivers **12-5**
Device Objects for VXIplug&play and IVI Drivers **12-6**
Connecting the Device Object **12-6**

Communicating with Instruments **12-8**
Configuring Instrument Settings **12-8**
Calling Device Object Methods **12-9**
Control Commands **12-10**

Device Groups **12-12**
Working with Group Objects **12-12**
Using Device Groups to Access Instrument Data **12-12**

13 Using VXIplug&play Drivers

VXIplug&play Setup **13-2**
Instrument Control Toolbox Software and VXIplug&play Drivers **13-2**
VISA Setup **13-2**
Other Software Requirements **13-2**

VXIplug&play Drivers **13-3**
Installing VXIplug&play Drivers **13-3**
Creating a MATLAB VXIplug&play Instrument Driver **13-3**
Constructing Device Objects Using a MATLAB VXIplug&play Instrument Driver **13-6**
Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI **13-6**

14 Using IVI Drivers

IVI Drivers Overview **14-2**
Instrument Control Toolbox Software and IVI Drivers **14-2**
IVI-C **14-2**

Instrument Interchangeability **14-3**
Minimal Code Changes **14-3**
Effective Use of Interchangeability **14-3**
Examples of Interchangeability **14-3**

Getting Started with IVI Drivers **14-5**
Introduction **14-5**
Requirements to Work with MATLAB **14-6**
Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI **14-8**
MATLAB IVI Instrument Driver **14-8**
Using MATLAB IVI Wrappers **14-10**

IVI Configuration Store **14-12**
Benefits of an IVI Configuration Store **14-12**
Components of an IVI Configuration Store **14-12**
Configuring an IVI Configuration Store **14-13**

Using IVI-C Class-Compliant Wrappers **14-17**
IVI-C Wrappers **14-17**
Prerequisites **14-17**
Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI **14-17**
Reading Waveforms Using the IVI-C Class Compliant Interface **14-18**
IVI-C Class Compliant Wrappers in Test & Measurement Tool **14-19**

The Quick-Control Interfaces **14-20**

Quick-Control Oscilloscope Requirements **14-21**

Read Waveforms Using the Quick-Control Oscilloscope **14-22**

Read a Waveform Using a Tektronix Scope **14-24**

Quick-Control Oscilloscope Functions **14-26**

Quick-Control Oscilloscope Properties **14-28**

Quick-Control Function Generator Requirements **14-30**

Generate Standard Waveforms Using the Quick-Control Function Generator **14-31**

Generate Arbitrary Waveforms Using Quick-Control Function Generator **14-33**

Quick-Control Function Generator Functions **14-35**

Quick-Control Function Generator Properties **14-37**

Quick-Control RF Signal Generator Requirements **14-40**

Quick-Control RF Signal Generator Functions **14-41**

Quick-Control RF Signal Generator Properties **14-43**

Download and Generate Signals with RF Signal Generator **14-45**
Create an RF Signal Generator Object **14-45**
Download a Waveform **14-46**
Generate Signal and Modulation Output **14-47**

Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI **14-48**

15 Instrument Support Packages

Install the Ocean Optics Spectrometers Support Package **15-2**

Install the NI-SCOPE Oscilloscopes Support Package **15-4**

Install the NI-FGEN Function Generators Support Package **15-5**

Install the NI-DCPower Power Supplies Support Package **15-6**

Install the NI-DMM Digital Multimeters Support Package **15-7**

Install the NI-845x I2C/SPI Interface Support Package **15-8**

Install the Total Phase Aardvark I2C/SPI Interface Support Package . . **15-9**

Install the NI-Switch Hardware Support Package **15-10**

Install the National Instruments VISA and ICP Interfaces Support Package **15-11**

Install the Keysight IO Libraries and VISA Interface Support Package **15-12**

16 Using Generic Instrument Drivers

Generic Drivers: Overview **16-2**

Writing a Generic Driver **16-3**
Creating the Driver and Defining Its Initialization Behavior **16-3**

Defining Properties . . . 16-4
Defining Functions . . . 16-6

Using Generic Driver with Test & Measurement Tool . . . **16-7**
Creating and Connecting the Device Object . . . 16-7
Accessing Properties . . . 16-8
Using Functions . . . 16-8

Using a Generic Driver at Command Line . . . **16-9**
Creating and Connecting the Device Object . . . 16-9
Accessing Properties . . . 16-9
Using Functions . . . 16-10

17 Saving and Loading the Session

Saving and Loading Instrument Objects . . . **17-2**
Saving Instrument Objects to a File . . . 17-2
Saving Objects to a MAT-File . . . 17-3

Debugging: Recording Information to Disk . . . **17-5**
Using the record Function . . . 17-5
Introduction to Recording Information . . . 17-5
Creating Multiple Record Files . . . 17-6
Specifying a File Name . . . 17-6
Record File Format . . . 17-6
Recording Information to Disk . . . 17-7

18 Test & Measurement Tool

Test & Measurement Tool Overview . . . **18-2**
Instrument Control Toolbox Software Support . . . 18-2
Navigating the Tree . . . 18-2

Using the Test & Measurement Tool . . . **18-4**
Overview of the Examples . . . 18-4
Hardware . . . 18-4
Instrument Objects . . . 18-9
Instrument Drivers . . . 18-13

19 Using the Instrument Driver Editor

MATLAB Instrument Driver Editor Overview . . . **19-2**
What Is a MATLAB Instrument Driver? . . . 19-2

How Does a MATLAB Instrument Driver Work? . . . 19-3
Why Use a MATLAB Instrument Driver? . . . 19-3

Creating MATLAB Instrument Drivers . . . **19-4**
Driver Components . . . 19-4
MATLAB Instrument Driver Editor Features . . . 19-4
Saving MATLAB Instrument Drivers . . . 19-5
Driver Summary and Common Commands . . . 19-5
Initialization and Cleanup . . . 19-8

Properties . . . **19-13**
Properties: Overview . . . 19-13
Property Components . . . 19-13
Examples of Properties . . . 19-15

Functions . . . **19-25**
Understanding Functions . . . 19-25
Function Components . . . 19-25
Examples of Functions . . . 19-26

Groups . . . **19-33**
Group Components . . . 19-33
Examples of Groups . . . 19-33

Using Existing Drivers . . . **19-47**
Modifying MATLAB Instrument Drivers . . . 19-47
Importing VXIplug&play and IVI Drivers . . . 19-47

20 Using the Instrument Driver Testing Tool

Instrument Driver Testing Tool Overview . . . **20-2**
Functionality . . . 20-2
Drivers . . . 20-2
Test Structure . . . 20-2
Starting . . . 20-3
Example . . . 20-3

Setting Up Your Test . . . **20-4**
Test File . . . 20-4
Providing a Name and Description . . . 20-4
Specifying the Driver . . . 20-4
Specifying an Interface . . . 20-4
Setting Test Preferences . . . 20-4
Setting Up a Driver Test . . . 20-5

Defining Test Steps . . . **20-9**
Test Step: Set Property . . . 20-9
Test Step: Get Property . . . 20-11
Test Step: Properties Sweep . . . 20-13
Test Step: Function . . . 20-15

Saving Your Test 20-19
Saving the Test as MATLAB Code 20-19
Saving the Test as a Driver Function 20-19

Testing and Results 20-21
Running All Steps 20-21
Partial Testing 20-22
Exporting Results 20-22
Saving Results 20-23

21 Instrument Control Toolbox Troubleshooting

How to Use This Troubleshooting Guide 21-3

Is My Hardware Supported? 21-4
Supported Interfaces 21-4
Supported Hardware 21-5

Troubleshooting SPI Interface 21-6
Supported Platforms 21-6
Adaptor Requirements 21-6
Configuration and Connection 21-7

Troubleshooting I2C Interface 21-10
Supported Platforms 21-10
Adaptor Requirements 21-10
Configuration and Connection 21-11

Troubleshooting MODBUS Interface 21-14
Supported Platforms 21-14
Configuration and Connection 21-14
Other Troubleshooting Tips for MODBUS 21-15

Troubleshooting Serial Port Interface 21-16
Issue 21-16
Possible Solutions 21-16

Troubleshooting GPIB Interface 21-19
Issue 21-19
Possible Solutions 21-19

Troubleshooting TCP/IP Client Interface 21-23
Issue 21-23
Possible Solutions 21-23

Troubleshooting TCP/IP Server Interface 21-25
Issue 21-25
Possible Solutions 21-25

Troubleshooting UDP Interface 21-27
Issue 21-27

Possible Solutions 21-27

Troubleshooting IVI and Quick-Control Interfaces **21-29**
Supported Platforms 21-29
Adaptor Requirements 21-29
Configuration and Connection 21-31

Troubleshooting VISA Interface **21-33**
Issue 21-33
Possible Solutions 21-33

Hardware Support Packages **21-36**

Deploying Standalone Applications with Instrument Control Toolbox **21-38**
Tips for both interface based communication and driver-based communication 21-38
Tips for interface based communication 21-38
Tips for driver based communication 21-38
Hardware Support packages 21-40

Contact MathWorks and Use the instrsupport Function **21-41**

Serial Warning - Unable to Read Any Data **21-42**

Serial Warning - Unable to Read All Data **21-43**

TCP/IP Warning - Unable to Read Any Data **21-45**

TCP/IP Warning - Unable to Read All Data **21-46**

UDP Warning - Unable to Read Any Data **21-48**

UDP Warning - Unable to Read All Data **21-50**

GPIB Warning - Unable to Read Any Data **21-52**

GPIB Warning - Unable to Read All Data **21-53**

TCP/IP Socket Using VISA Warning - Unable to Read Any Data **21-55**

TCP/IP Socket Using VISA Warning - Unable to Read All Data **21-56**

Bluetooth Warning - Unable to Read Any Data **21-58**

Bluetooth Warning - Unable to Read All Data **21-59**

Serialport Warning - Unable to Read Any Data **21-61**

Serialport Warning - Unable to Read All Data **21-62**

Resolve TCP/IP Client Warning: Unable to Read Any Data **21-63**
Issue 21-63
Possible Solutions 21-63

Resolve TCP/IP Server Warning: Unable to Read Any Data **21-64**
Issue **21-64**
Possible Solutions **21-64**

Resolve TCP/IP Server Warning: Unable to Read All Data **21-65**
Issue **21-65**
Possible Solutions **21-65**

Resolve UDP Port Warning: Unable to Read Any Data **21-66**
Issue **21-66**
Possible Solutions **21-66**

Resolve UDP Port Warning: Unable to Read All Data **21-67**
Issue **21-67**
Possible Solutions **21-67**

Resolve VISA Warning: Unable to Read Any Data **21-68**
Issue **21-68**
Possible Solutions **21-68**

Resolve VISA Warning: Unable to Read All Data **21-70**
Issue **21-70**
Possible Solutions **21-70**

Resolve Serial Port Connection Errors **21-71**
Issue **21-71**
Possible Solutions **21-71**

Resolve TCP/IP Client Connection Errors **21-73**
Issue **21-73**
Possible Solutions **21-73**

Resolve TCP/IP Server Connection Errors **21-75**
Issue **21-75**
Possible Solutions **21-75**

Resolve UDP Port Connection Errors **21-76**
Issue **21-76**
Possible Solutions **21-76**

Resolve VISA Connection Errors **21-77**
Issue **21-77**
Possible Solutions **21-77**

22 Instrument Control Toolbox Examples

Recording an Instrument Session **22-3**

Reading Waveforms from an Oscilloscope Using a Quick-Control Oscilloscope Object **22-9**

Creating and Downloading an Arbitrary Waveform to a Function Generator **22-12**

Restoring the BlinkM to Its Factory Settings Using I2C Bus **22-15**

Communicating with EEPROM using SPI bus **22-18**

Communicating with the Lego® Mindstorms® NXT brick over Bluetooth® **22-20**

Tap Detection with ADXL345 Accelerometer Chip Using the NI USB 8451 Adaptor **22-24**

Reading Inphase and Quadrature (IQ) Data from a Signal Analyzer over TCP/IP **22-29**

Fetch Waveform through NI-SCOPE MATLAB Instrument Driver in Simulation Mode **22-35**

Using NI-FGEN Instrument Driver To Generate A Sine Wave **22-39**

Read Waveform Data from Keysight® DSO-X 2002A Oscilloscopes Using the IVI-C Driver **22-41**

Read Waveforms from a Keysight® M9210A Digitizer using the IVI-C Driver **22-44**

Set Output Voltage and Make Measurements on Keysight® AC6801A Power Supply Using the IVI-C Driver **22-48**

Measure Frequency on Keysight® 532xx Frequency Counter Using the IVI-C Driver **22-54**

Measure AC Voltage on a Keysight® 34410A Digital Multimeter Using the IVI-C Driver **22-59**

Set Output Voltage and Make Measurements from a Keysight® AgE3633A DC Power Supply Using the IVI-C Driver **22-62**

Generate AM Waveforms on Keysight® 3352x Waveform Generator Using the IVI-C Driver **22-65**

Measure Power on a Keysight® RF Power Meter Using the IVI-C Driver **22-68**

Configure Output Signal on Keysight® RF Signal Generator Using the IVI-C Driver **22-71**

Set and Measure DAC (Data Acquisition) Channel Voltage on Keysight® 34970A Switch Using the IVI-C Driver **22-74**

Acquire Signal Spectrum on a Rohde & Schwarz® Spectrum Analyzer Using the IVI-C Driver **22-77**

Basic UDP Communication **22-81**

Video Surveillance Over TCP/IP Network **22-83**

802.11 OFDM Beacon Frame Generation and Transmission with Test and Measurement Equipment **22-85**

LTE Waveform Generation and Transmission Using Quick Control RF Signal Generator **22-89**

Creating and Downloading an IQ Waveform to a RF Signal Generator **22-93**

Fetch Spectrum Through Ocean Optics Spectrometer Using MATLAB Instrument Driver **22-96**

Read Streaming Data from Arduino Using Serial Port Communication **22-99**

Read Waveform from Tektronix TDS 1002 Scope Using SCPI Commands **22-102**

Read Voltage Through NI-DMM MATLAB Instrument Driver in Simulation Mode **22-105**

Using a NI Switch Module and a NI DMM to Perform Resistance Measurements **22-108**

Generate DC Voltage Using NI-DCPOWER MATLAB Instrument Driver in Simulation Mode **22-112**

Send and Receive Multicast Data Packets Using the User Datagram Protocol **22-115**

Communicate Between Two MATLAB Sessions Using User Datagram Protocol **22-118**

Broadcast User Datagram Protocol Data Packets **22-122**

Communicate Binary and ASCII Data to an Echo Server Using TCP/IP **22-124**

Communicate Between a TCP/IP Client and Server in MATLAB **22-127**

Read Data from Arduino Using TCP/IP Communication **22-132**

Generate a Swept Sinusoid Using VISA and Capture Waveform Using Quick-Control Oscilloscope **22-136**

5G NR Waveform Acquisition and Analysis **22-140**

TCP/IP Client Block Communication with Arduino Server **22-152**

23 Using the Instrument Control Toolbox Block Library

Open Instrument Control Toolbox Block Library **23-2**
From the Command Line **23-2**
From the Simulink Library Browser **23-2**

Send and Receive Data Through Serial Port Loopback **23-5**
Step 1: Create a New Model **23-5**
Step 2: Open the Block Library **23-5**
Step 3: Drag the Instrument Control Toolbox Blocks into the Model **23-6**
Step 4: Drag Other Blocks to Complete the Model **23-7**
Step 5: Connect the Blocks **23-8**
Step 6: Specify the Block Parameter Values **23-9**
Step 7: Specify the Block Priority **23-12**
Step 8: Run the Simulation **23-12**

Send and Receive Data over TCP/IP Network **23-14**
Step 1: Create an Echo Server **23-14**
Step 2: Create a New Model **23-14**
Step 3: Open the Block Library **23-15**
Step 4: Drag the Instrument Control Toolbox Blocks into the Model . . . **23-16**
Step 5: Drag the Sine Wave and Scope Blocks to Complete the Model . . **23-16**
Step 6: Connect the Blocks **23-18**
Step 7: Specify the Block Parameter Values **23-19**
Step 8: Specify the Block Priorities **23-21**
Step 9: Run the Simulation **23-22**
Step 10: View the Result **23-23**

Enable Blocking Mode in Receive and Send Blocks **23-25**
Blocking Mode **23-25**
Nonblocking Mode **23-26**

Timing in Hardware Interface Models **23-30**
Simulation Time **23-30**
Block Sample Time **23-30**
Pacing Model Simulation **23-31**

24 Functions

25 Properties

26 Blocks

A Bibliography

Getting Started

- "Instrument Control Toolbox Product Description" on page 1-2
- "Instrument Control Toolbox Overview" on page 1-3
- "About Instrument Control" on page 1-5
- "Installation Information" on page 1-8
- "Supported Hardware" on page 1-9
- "Examining Your Hardware Resources" on page 1-10
- "Communicating with Your Instrument" on page 1-16
- "General Preferences for Instrument Control" on page 1-20
- "Interface and Property Help" on page 1-24

Instrument Control Toolbox Product Description

Control test and measurement instruments and communicate with computer peripherals and industrial automation equipment

Instrument Control Toolbox lets you connect MATLAB® directly to instruments such as oscilloscopes, function generators, signal analyzers, power supplies, and analytical instruments. The toolbox connects to your instruments via instrument drivers such as IVI and VXI*plug&play*, or via text-based SCPI commands over commonly used communication protocols such as GPIB, VISA, TCP/IP, and UDP. You can also control and acquire data from test equipment without writing code.

With Instrument Control Toolbox, you can generate data in MATLAB to send out to an instrument or read data into MATLAB for analysis and visualization. You can automate tests, verify hardware designs, and build test systems based on LXI, PXI, and AXIe standards.

The toolbox provides built-in support for TCP/IP, UDP, I2C, SPI, and Bluetooth® serial protocols for remote communication with other computers and printed circuit boards (PCBs) from MATLAB. It also includes functions and apps for the MODBUS® protocol, enabling communication with industrial automation equipment such as programmable logic controllers (PLCs) and programmable automation controllers (PACs).

Instrument Control Toolbox Overview

In this section...

"Getting to Know the Instrument Control Toolbox Software" on page 1-3
"Exploring the Instrument Control Toolbox Software" on page 1-3
"Learning About the Instrument Control Toolbox Software" on page 1-4
"Using the Documentation Examples" on page 1-4

Getting to Know the Instrument Control Toolbox Software

Instrument Control Toolbox software is a collection of MATLAB functions built on the MATLAB technical computing environment. The toolbox provides you with these features:

- A framework for communicating with instruments that support the GPIB interface (IEEE®-488), the VISA standard, and the TCP/IP and UDP protocols. Note that the toolbox extends the basic serial port features included with the MATLAB software.
- Support for IVI®, VXI*plug&play*, and MATLAB instrument drivers.
- Functions for transferring data between the MATLAB workspace and your instrument:
  - The data can be binary (numerical) or text.
  - The transfer can be synchronous and block access to the MATLAB Command Window, or asynchronous and allow access to the MATLAB Command Window.
- Event-based communication.
- Functions for recording data and event information to a text file.
- Tools that facilitate instrument control in an easy-to-use graphical environment.

Instrument Control Toolbox provides access to Keysight™ Command Expert from MATLAB to control and script instrument actions. In addition, Keysight Command Expert generates MATLAB code that can be used from Instrument Control Toolbox. To learn more, see the documentation for Keysight Command Expert version 1.1 or later, or

`https://www.mathworks.com/agilentcmdexpert`

Exploring the Instrument Control Toolbox Software

For a list of the toolbox functions, type

```
help instrument
```

For the code of a function, type

```
type function_name
```

For help for any function, type

```
instrhelp function_name
```

You can change the way any toolbox function works by copying and renaming the file, then modifying your copy. You can also extend the toolbox by adding your own files, or by using it in combination with other products such as MATLAB Report Generator™or Data Acquisition Toolbox™ product.

To use the Instrument Control Toolbox product, you should be familiar with the:

- Basic features of MATLAB.
- Appropriate commands used to communicate with your instrument. These commands might use the SCPI language or they might be methods associated with an IVI, VXI*plug&play*, or MATLAB instrument driver.
- Features of the interface associated with your instrument.

Learning About the Instrument Control Toolbox Software

Start with this set of topics, which describe how to examine your hardware resources, how to communicate with your instrument, how to get online help, and so on. Then click on the **Getting Started** link at the top of the page and read the topics contained there, which provide a framework for constructing instrument control applications. Depending on the interface used by your instrument, you might then want to read the appropriate interface-specific chapter.

If you want detailed information about a specific function, refer to the functions documentation. If you want detailed information about a specific property, refer to the properties documentation.

Using the Documentation Examples

The examples in this guide use specific instruments such as a Tektronix® TDS 210 two-channel oscilloscope or a Keysight 33120A function generator. Additionally, the GPIB examples use a National Instruments® GPIB controller and the serial port examples use the Windows® specific COM1 serial port. The string commands written to these instruments are often unique to the vendor, and the address information such as the board index or primary address associated with the hardware reflects a specific configuration.

These examples appear throughout the documentation. You should modify the examples to work with your specific hardware configuration.

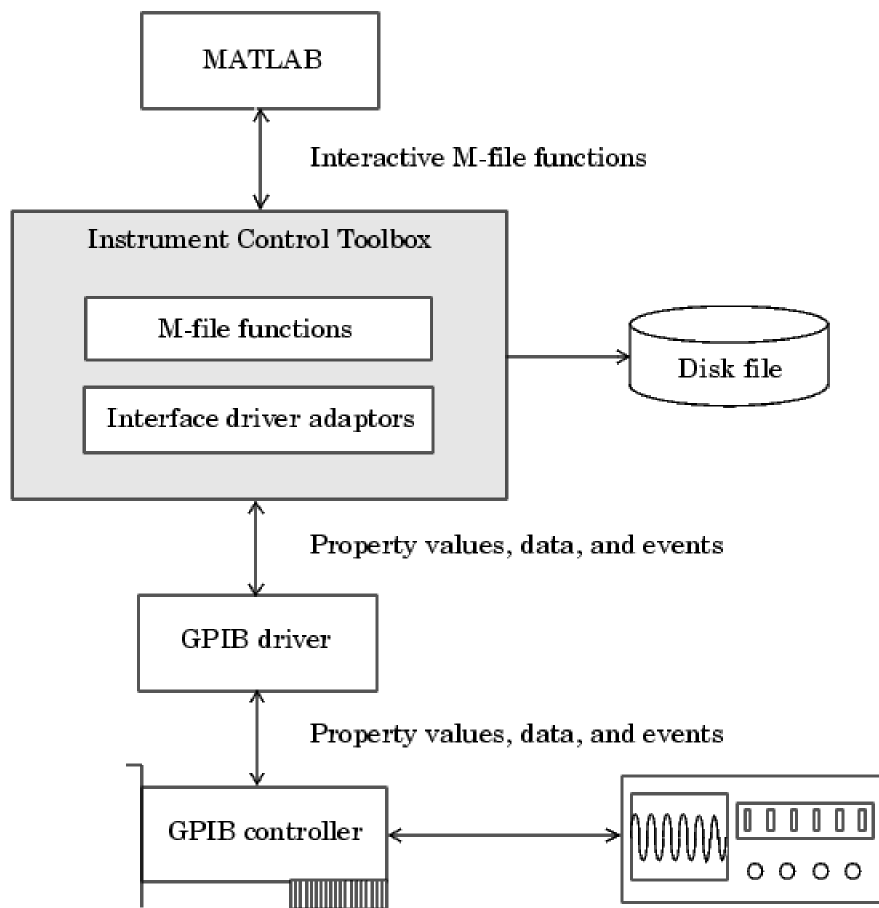
About Instrument Control

In this section...

"Passing Information Between the MATLAB Workspace and Your Instrument" on page 1-5
"MATLAB Functions" on page 1-6
"Interface Driver Adaptor" on page 1-6

Passing Information Between the MATLAB Workspace and Your Instrument

Instrument Control Toolbox software consists of two distinct components: MATLAB functions and interface driver adaptors. These components allow you to pass information between the MATLAB workspace and your instrument. For example, the following diagram shows how information passes from the MATLAB software to an instrument via the GPIB driver and the GPIB controller.



This diagram illustrates how information flows from component to component. Information consists of

- **Property values**

  You define the behavior of your instrument control application by configuring property values. In general, you can think of a property as a characteristic of the toolbox or of the instrument that can be configured to suit your needs.
- **Data**

  You can write data to the instrument and read data from the instrument. Data can be binary (numerical) or formatted as text. Writing text often involves writing string commands that change hardware settings, or prepare the instrument to return data or status information, while writing binary data involves writing numerical values such as calibration or waveform data.
- **Events**

  An event occurs after a condition is met and might result in one or more callbacks. Events can be generated only after you configure the associated properties. For example, you can use events to analyze data after a certain number of bytes are read from the instrument, or display a message to the MATLAB command line after an error occurs.

MATLAB Functions

To perform any task within your instrument control application, you must call MATLAB functions from the MATLAB workspace. Among other things, these functions allow you to:

- Create instrument objects, which provide a gateway to your instrument's capabilities and allow you to control the behavior of your application.
- Connect the object to the instrument.
- Configure property values.
- Write data to the instrument, and read data from the instrument.
- Examine your hardware resources and evaluate your application status.

For a listing of all Instrument Control Toolbox software functions, refer to the functions documentation. You can also display the toolbox functions by typing

```
help instrument
```

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Interface Driver Adaptor

The interface driver adaptor (or just *adaptor*) is the link between the toolbox and the interface driver. The adaptor's main purpose is to pass information between the MATLAB workspace and the interface driver. Interface drivers are provided by your instrument vendor. For example, if you are communicating with an instrument using a National Instruments GPIB controller, then an interface driver such as NI-488.2 must be installed on your platform. Note that interface drivers are not installed as part of the Instrument Control Toolbox software.

Instrument Control Toolbox software provides adaptors for the GPIB interface and the VISA standard. The serial port, TCP/IP, and UDP interfaces do not require an adaptor.

Interface Adaptors

Interface	Adaptor Name
GPIB	`keysight` (note that `agilent` still also works), `ics`, `mcc`, `ni`
Serial port	N/A
TCP/IP	N/A
UDP	N/A
VISA standard	`keysight` (note that `agilent` still also works), `ni`, `rs`, `tek`

As described in "Examining Your Hardware Resources" on page 1-10, you can list the supported interfaces and adaptor names with the `instrhwinfo` function.

Installation Information

In this section...

"Installation Requirements" on page 1-8
"Toolbox Installation" on page 1-8
"Hardware and Driver Installation" on page 1-8

Installation Requirements

To communicate with your instrument from the MATLAB workspace, you must install these components:

- MATLAB
- Instrument Control Toolbox software

Additionally, you might need to install hardware such as a GPIB controller and vendor-specific software such as drivers, support libraries, and so on. For a complete list of all supported vendors, refer to "Interface Driver Adaptor" on page 1-6.

Toolbox Installation

To determine if Instrument Control Toolbox software is installed on your system, type

```
ver
```

at the MATLAB Command Window. The MATLAB Command Window displays information about the version of the MATLAB software you are running, including a list of installed add-on products and their version numbers. Check the list to see if Instrument Control Toolbox appears.

For information about installing the toolbox, refer to the installation documentation for your platform. If you experience installation difficulties, look for the installation and license information at the MathWorks® Web site (`https://www.mathworks.com/support`).

Hardware and Driver Installation

Installation of hardware devices such as GPIB controllers, instrument drivers, support libraries, and so on is described in the documentation provided by the instrument vendor. Many vendors provide the latest drivers through their Web site.

Note You must install all necessary device-specific software provided by the instrument vendor in addition to the Instrument Control Toolbox software.

Supported Hardware

The following table lists the hardware support for the Instrument Control Toolbox. Notes follow the table.

Feature	64-bit MATLAB on Windows	64-bit MATLAB on macOS	64-bit MATLAB on Linux®
Serial	supported	supported	supported
TCP/IP	supported	supported	supported
UDP	supported	supported	supported
VISA [3]	supported [1]	supported on two vendors [1, 3]	
GPIB [4]	supported [1]		
I2C [5]	supported [1]	supported [1]	supported [1]
SPI [5]	supported [1]	supported [1]	supported [1]
MODBUS	supported	supported	supported
Quick-Control Oscilloscope and Quick-Control Function Generator	supported [2]	supported [2]	supported [2]
MATLAB Instrument Drivers	supported	supported	supported
MATLAB Instrument Drivers made using IVI-C drivers and Instrument Wrappers for IVI-C drivers	supported [1]		

Table Notes

1. Dependent on support by third-party vendor driver for the hardware on this platform.

2. Dependent on third-party vendor support of platform when using an IVI-driver with Quick-Control Oscilloscope or Quick-Control Function Generator.

3. Requires Keysight (formerly Agilent®), National Instruments, Rohde & Schwartz, or TAMS VISA compliant with VISA specification 5.0 or higher for any platform. Only National Instruments VISA and Rohde & Schwartz VISA are supported on macOS. The other vendors' VISA support does not include macOS.

4. Requires Keysight (formerly Agilent), ICS Electronics™, Measurement Computing™ (MCC), ADLINK Technology, or National Instruments hardware and driver.

5. Requires Aardvark or National Instruments hardware and driver.

Examining Your Hardware Resources

In this section...

"instrhwinfo Function" on page 1-10
"Test & Measurement Tool" on page 1-13
"Viewing the IVI Configuration Store" on page 1-14

instrhwinfo Function

You can examine the hardware-related resources visible to the toolbox with the `instrhwinfo` function. The specific information returned by `instrhwinfo` depends on the supplied arguments, and is divided into these categories:

- "General Toolbox Information" on page 1-10
- "Interface Information" on page 1-10
- "Adaptor Information" on page 1-11
- "Instrument Object Information" on page 1-12
- "Installed Driver Information" on page 1-12

General Toolbox Information

For general information about the Instrument Control Toolbox, type:

```
instrhwinfo

        MATLABVersion: '7.0 (R14)'
  SupportedInterfaces: {'gpib'  'serial'  'visa'  'tcpip'  'udp'}
     SupportedDrivers: {'matlab'  'vxipnp'  'ivi'}
          ToolboxName: 'Instrument Control Toolbox'
       ToolboxVersion: '2.0 (R14)'
```

The `SupportedInterfaces` and `SupportedDrivers` fields list the interfaces and drivers supported by the toolbox, and not necessarily those installed on your computer.

Interface Information

To display information about a specific interface, you supply the interface name as an argument to `instrhwinfo`. The interface name can be `gpib`, `serial`, `tcpip`, `udp`, or `visa`.

For the GPIB and VISA interfaces, the information includes installed adaptors. For the serial port interface, the information includes the available ports. For the TCP/IP and UDP interfaces, the information includes the local host address. For example, to display the GPIB interface information:

```
out = instrhwinfo('gpib')
out =

    InstalledAdaptors: {'ics' 'ni'}
       JarFileVersion: 'Version 2.0 (R14)'
```

The `InstalledAdaptors` field indicates that ICS Electronics (ICS) and National Instruments drivers are installed. Therefore, you can communicate with instruments using GPIB controllers from these vendors.

Adaptor Information

To display information about a specific installed adaptor, you supply the interface name and the adaptor name as arguments to `instrhwinfo`.

Interface Name	Adaptor Name
`gpib`	`keysight` (note that `agilent` still also works), `ics`, `mcc`, `adlink`, `ni`
`visa`	`keysight` (note that `agilent` still also works), `ni`, `rs`, `tek`

The returned information describes the adaptor, the vendor driver, and the object constructors. For example, to display information for the National Instruments GPIB adaptor,

```
ghwinfo = instrhwinfo('gpib','ni')

ghwinfo = 

             AdaptorDllName: [1x82 char]
          AdaptorDllVersion: 'Version 2.0 (R14)'
                AdaptorName: 'NI'
          InstalledBoardIds: 0
      ObjectConstructorName: {'gpib('ni', 0, 2);'}
              VendorDllName: 'gpib-32.dll'
    VendorDriverDescription: 'NI-488'
```

The `ObjectConstructorName` field provides the syntax for creating a GPIB object for the National Instruments adaptor. In this example, the GPIB controller has board index `0` and the instrument has primary address `2`.

```
g = gpib('ni',0,2);
```

To display information for the Tektronix VISA adaptor,

```
vhwinfo = instrhwinfo('visa','tek')
vhwinfo = 

             AdaptorDllName: [1x83 char]
          AdaptorDllVersion: 'Version 2.0 (R14 Beta 1)'
                AdaptorName: 'TEK'
           AvailableChassis: []
       AvailableSerialPorts: {2x1 cell}
          InstalledBoardIds: 0
      ObjectConstructorName: {3x1 cell}
                SerialPorts: {2x1 cell}
              VendorDllName: 'visa32.dll'
    VendorDriverDescription: 'Tektronix VISA Driver'
        VendorDriverVersion: 2.0500
```

The available VISA object constructor names are shown below.

```
vhwinfo.ObjectConstructorName
ans = 

    'visa('tek', 'ASRL1::INSTR');'
    'visa('tek', 'ASRL2::INSTR');'
    'visa('tek', 'GPIB0::1::INSTR');'
```

The `ObjectConstructorName` field provides the syntax for creating a VISA object for the GPIB and serial port interfaces. In this example, the GPIB controller has board index `0` and the instrument has primary address `1`.

```
vg = visa('tek','GPIB0::1::INSTR');
```

Instrument Object Information

To display information about a specific instrument object, you supply the object as an argument to `instrhwinfo`. For example, to display information for the GPIB object created in the ("Adaptor Information" on page 1-11), type:

```
ghwinfo = instrhwinfo(g)
ghwinfo =

             AdaptorDllName: [1x82 char]
          AdaptorDllVersion: 'Version 2.0 (R14)'
                AdaptorName: 'NI'
              VendorDllName: 'gpib-32.dll'
    VendorDriverDescription: 'NI-488'
```

To display information for the VISA-GPIB object created in the ("Adaptor Information" on page 1-11), type:

```
vghwinfo = instrhwinfo(vg)
vghwinfo =

             AdaptorDllName: [1x83 char]
          AdaptorDllVersion: 'Version 2.0 (R14)'
                AdaptorName: 'TEK'
              VendorDllName: 'visa32.dll'
    VendorDriverDescription: 'Tektronix VISA Driver'
        VendorDriverVersion: 2.0500
```

Alternatively, you can return hardware information via the Workspace browser by right-clicking an instrument object, and selecting **Display Hardware Info** from the context menu.

Installed Driver Information

To display information about a supported driver type, you supply the driver type as an argument to `instrhwinfo`. For example, to display information for the IVI configuration, type:

```
instrhwinfo('ivi')
ans =
             LogicalNames: {'MyIviCLogical'  'MyScope'  'TekScope'}
               ProgramIDs: {'TekScope.TekScope'}
                  Modules: {'ag3325b'}
ConfigurationServerVersion: '1.3.1.0'
  MasterConfigurationStore: 'D:\Apps\IVI\Data\IviConfigurationStore.xml'
              IVIRootPath: 'D:\Apps\IVI\'
```

To display information about a specific driver or resource, you supply the driver name in addition to the type as an argument to `instrhwinfo`. For example, to display information about the `ag3325b` VXI*plug&play* driver:

```
instrhwinfo('vxipnp', 'ag3325b')
ans =
     Manufacturer: 'Agilent Technologies'
            Model: 'Agilent 3325B Synthesizer/Func. Gen.'
```

```
    DriverVersion: '4.1'
    DriverDllName: 'C:\VXIPNP\WINNT\bin\ag3325b_32.dll'
```

Test & Measurement Tool

You can use the Test & Measurement Tool (`tmtool`) to manage the resources of your instrument control session. You can use this tool to:

- Search for installed adaptors.
- Examine available hardware.
- Examine installed drivers.
- Examine instrument objects.

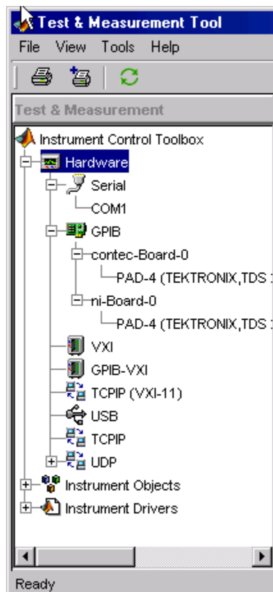
To open the Test & Measurement Tool, type:

```
tmtool
```

Hardware

Expand the `Hardware` node in the tree to list the supported interfaces.

Right-click the `Hardware` node to scan for instrument hardware. The interface nodes expand to include entries for each instrument found by the scan.

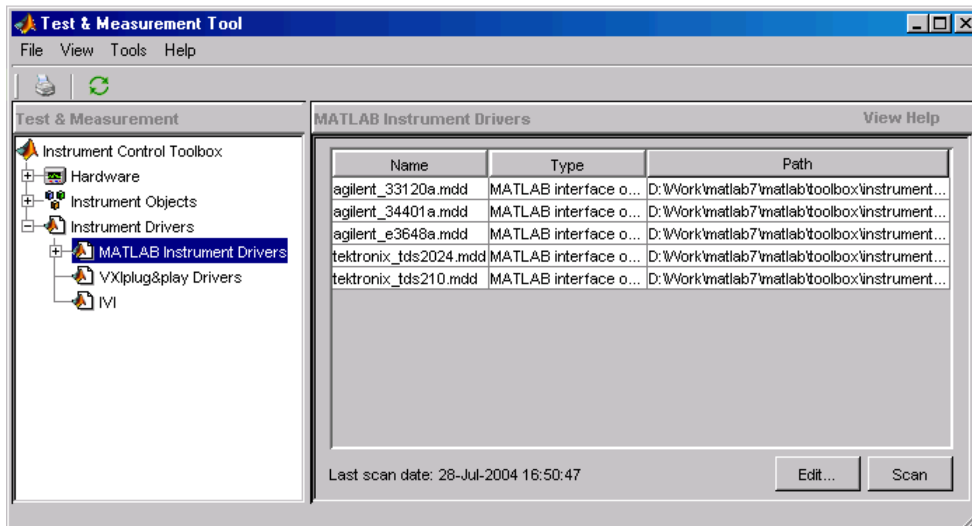


Installed Drivers

The Test & Measurement Tool can display your installed drivers. The three categories of drivers are `MATLAB Instrument Drivers`, `VXIplug&play Drivers`, and `IVI`, as shown below under the expanded `Instrument Drivers` node.

Right-click the `Instrument Drivers` node to scan for installed drivers. The driver-type nodes expand to include entries for each driver found by the scan. Note that for MATLAB instrument drivers and VXI*plug&play* drivers, the installation of a driver requires only the presence of a driver file. For

IVI, installation involves an IVI configuration store; see “Viewing the IVI Configuration Store” on page 1-14.



The Test & Measurement Tool GUI includes embedded help. For further details about the Test & Measurement Tool and its capabilities, see “Test & Measurement Tool Overview” on page 18-2.

Viewing the IVI Configuration Store

An IVI configuration store greatly enhances instrument interchangeability by providing the means to configure the relationship between drivers and I/O interface references outside of the application. For details of the components of an IVI configuration store, see “IVI Configuration Store” on page 14-12.

Command-Line Configuration

You can use command-line functions to examine and configure your IVI configuration store. To see what IVI configuration store elements are available, use `instrhwinfo` to identify the existing logical names.

```
instrhwinfo('ivi')
ans =
              LogicalNames: {'MainScope', 'FuncGen'}
                ProgramIDs: {'TekScope.TekScope','Agilent33250'}
                   Modules: {'ag3325b', 'hpe363xa'}
ConfigurationServerVersion: '1.3.1.0'
  MasterConfigurationStore: 'C:\Program Files\IVI\Data\
                             IviConfigurationStore.xml'
               IVIRootPath: 'C:\Program Files\IVI\'
```

Use `instrhwinfo` with a logical name as an argument to see the details of that logical name's configuration.

```
instrhwinfo('ivi','MainScope')
ans =
            DriverSession: 'TekScope.DriverSession'
            HardwareAsset: 'TekScope.Hardware'
           SoftwareModule: 'TekScope.Software'
     IOResourceDescriptor: 'GPIB0::13::INSTR'
```

```
SupportedInstrumentModels: 'TekScope 5000, 6000 and 7000 series'
        ModuleDescription: 'TekScope software module desc'
           ModuleLocation: ''
```

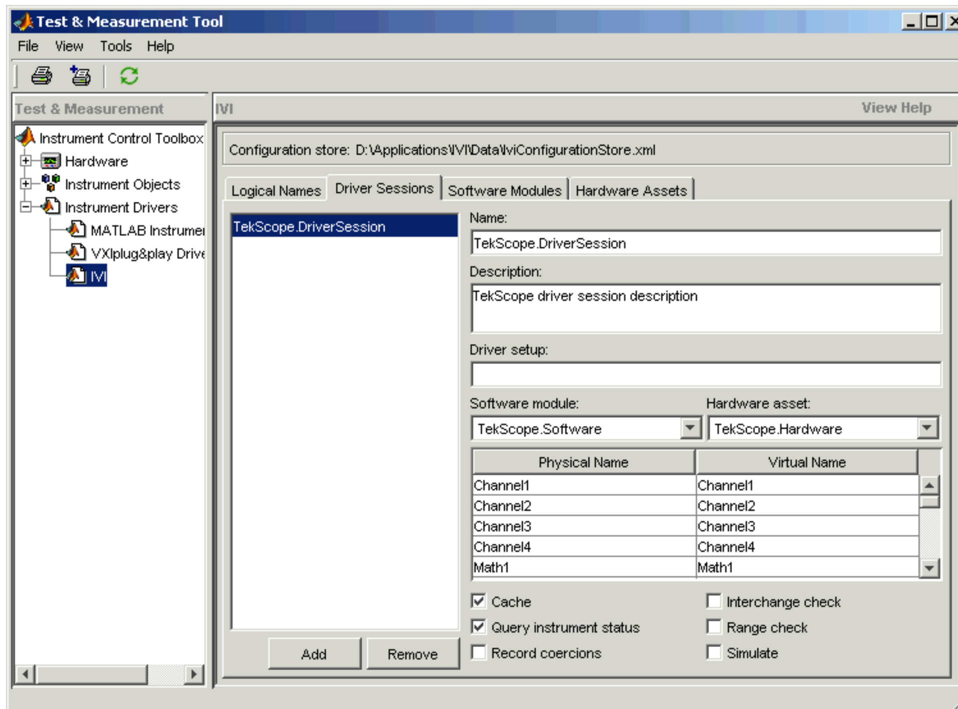
You create and configure elements in the IVI configuration store by using the IVI configuration store object functions `add`, `commit`, `remove`, and `update`. For further details, see the reference pages for these functions.

Using the Test & Measurement Tool

You can use the Test & Measurement Tool to examine or configure your IVI configuration store. To open the tool, type:

```
tmtool
```

Expand the `Instrument Drivers` node and click `IVI`.



You see a tab for each type of IVI configuration store element. This figure shows the available driver sessions in the current IVI configuration store. For the selected driver session, you can use any available software module or hardware asset. This figure shows the configuration for the driver session `TekScope.DriverSession`, which uses the software module `TekScope.Software` and the hardware asset `TekScope.Hardware`.

Communicating with Your Instrument

In this section...

"Instrument Control Session Examples" on page 1-16

"Communicating with a GPIB Instrument" on page 1-16

"Communicating with a GPIB-VXI Instrument" on page 1-17

"Communicating with a Serial Port Instrument" on page 1-17

"Communicating with a GPIB Instrument Using a Device Object" on page 1-18

Instrument Control Session Examples

Each example illustrates a typical *instrument control session*. The instrument control session comprises all the steps you are likely to take when communicating with a supported instrument. You should keep these steps in mind when constructing your own instrument control applications.

The examples also use specific instrument addresses, SCPI commands, and so on. If your instrument requires different parameters, or if it does not support the SCPI language, you should modify the examples accordingly. For more information, see Using SCPI Commands on page 3-9.

If you want detailed information about any functions that are used, refer to the functions documentation. If you want detailed information about any properties that are used, refer to the properties documentation.

Communicating with a GPIB Instrument

This example illustrates how to communicate with a GPIB instrument. The GPIB controller is a National Instruments AT-GPIB card. The instrument is a Keysight 33120A Function Generator, which is generating a 2 volt peak-to-peak signal.

You should modify this example to suit your specific instrument control application needs. If you want detailed information about communicating with an instrument via GPIB, refer to "GPIB Overview" on page 4-2.

1 **Create an interface object** — Create the GPIB object `g` associated with a National Instruments GPIB board with board index 0, and an instrument with primary address 1.

```
g = gpib('ni',0,1);
```

2 **Connect to the instrument** — Connect `g` to the instrument.

```
fopen(g)
```

3 **Configure property values** — Configure `g` to assert the EOI line when the line feed character is written to the instrument, and to complete read operations when the line feed character is read from the instrument.

```
g.EOSMode = 'read&write'
g.EOSCharCode = 'LF'
```

4 **Write and read data** — Change the instrument's peak-to-peak voltage to three volts by writing the `Volt 3` command, query the peak-to-peak voltage value, and then read the voltage value.

```
fprintf(g,'Volt 3')
fprintf(g,'Volt?')
```

```
data = fscanf(g)
data =
+3.00000E+00
```

5 **Disconnect and clean up** — When you no longer need `g`, you should disconnect it from the instrument, remove it from memory, and remove it from the MATLAB workspace.

```
fclose(g)
delete(g)
clear g
```

Communicating with a GPIB-VXI Instrument

This example illustrates how to communicate with a VXI instrument via a GPIB controller using the VISA standard provided by Keysight.

The GPIB controller is a Keysight E1406A command module in VXI slot 0. The instrument is a Keysight E1441A Function/Arbitrary Waveform Generator in VXI slot 1, which is outputting a 2 volt peak-to-peak signal. The GPIB controller communicates with the instrument over the VXI backplane.

You should modify this example to suit your specific instrument control application needs. If you want detailed information about communicating with an instrument using VISA, refer to “Get Started with VISA” on page 5-2.

1 **Create an instrument object** — Create the VISA-GPIB-VXI object `v` associated with the E1441A instrument located in chassis 0 with logical address 80.

```
v = visa('keysight','GPIB-VXI0::80::INSTR');
```

2 **Connect to the instrument** — Connect `v` to the instrument.

```
fopen(v)
```

3 **Configure property values** — Configure `v` to complete a read operation when the line feed character is read from the instrument.

```
v.EOSMode = 'read'
v.EOSCharCode = 'LF'
```

4 **Write and read data** — Change the instrument's peak-to-peak voltage to three volts by writing the `Volt 3` command, query the peak-to-peak voltage value, and then read the voltage value.

```
fprintf(v,'Volt 3')
fprintf(v,'Volt?')
data = fscanf(v)
data =
+3.00000E+00
```

5 **Disconnect and clean up** — When you no longer need `v`, you should disconnect it from the instrument, remove it from memory, and remove it from the MATLAB workspace.

```
fclose(v)
delete(v)
clear v
```

Communicating with a Serial Port Instrument

This example illustrates how to communicate with an instrument via the serial port. The instrument is a Tektronix TDS 210 two-channel digital oscilloscope connected to the serial port of a PC, and configured for a baud rate of 4800 and a carriage return (CR) terminator.

You should modify this example to suit your specific instrument control application needs. If you want detailed information about communicating with an instrument connected to the serial port, refer to “Serial Port Overview” on page 6-2.

Note This example is Windows specific.

1. **Create an instrument object** — Create the serial port object `s` associated with the COM1 serial port.

   ```
   s = serial('COM1');
   ```
2. **Configure property values** — Configure `s` to match the instrument's baud rate and terminator.

   ```
   s.BaudRate = 4800
   s.Terminator = 'CR'
   ```
3. **Connect to the instrument —** Connect `s` to the instrument. This step occurs after property values are configured because serial port instruments can transfer data immediately after the connection is established.

   ```
   fopen(s)
   ```
4. **Write and read data** — Write the `*IDN?` command to the instrument and then read back the result of the command. `*IDN?` queries the instrument for identification information.

   ```
   fprintf(s,'*IDN?')
   out = fscanf(s)
   out =
   TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
   ```
5. **Disconnect and clean up** — When you no longer need `s`, you should disconnect it from the instrument, remove it from memory, and remove it from the MATLAB workspace.

   ```
   fclose(s)
   delete(s)
   clear s
   ```

Communicating with a GPIB Instrument Using a Device Object

This example illustrates how to communicate with a GPIB instrument through a device object. The GPIB controller is a Measurement Computing card, and the instrument is a Keysight 33120A Function Generator, which you set to produce a 1 volt peak-to-peak sine wave at 1,000 Hz. Device objects use instrument drivers; this example uses the driver `agilent_33120a.mdd`.

You should modify this example to suit your specific instrument control application needs. If you want detailed information about communicating through device objects, see “Device Objects” on page 12-2.

1. **Create instrument objects** — Create the GPIB object `g` associated with a Measurement Computing GPIB board with board index 0, and an instrument with primary address 4. Then create the device object `d` associated with the interface object `g`, and with the instrument driver `agilent_33120a.mdd`.

   ```
   g = gpib('mcc',0,4);
   d = icdevice('agilent_33120a.mdd',g);
   ```
2. **Connect to the instrument** — Connect `d` to the instrument.

   ```
   connect(d)
   ```

3. **Call device object method** — Use the `devicereset` method to set the generator to a known configuration. The behavior of the generator for this method is defined in the instrument driver.

   ```
   devicereset(d)
   ```

4. **Configure property values** — Configure `d` to set the amplitude and frequency for the signal from the function generator.

   ```
   d.Amplitude = 1.00
   d.AmplitudeUnits = 'vpp'
   d.Frequency = 1000
   ```

5. **Disconnect and clean up** — When you no longer need `d` and `g`, you should disconnect from the instrument, remove the objects from memory, and remove them from the MATLAB workspace.

   ```
   disconnect(d)
   delete([d g])
   clear d g
   ```

General Preferences for Instrument Control

In this section...

"Accessing General Preferences" on page 1-20

"MATLAB Instrument Driver Editor" on page 1-21

"MATLAB Instrument Driver Testing Tool" on page 1-21

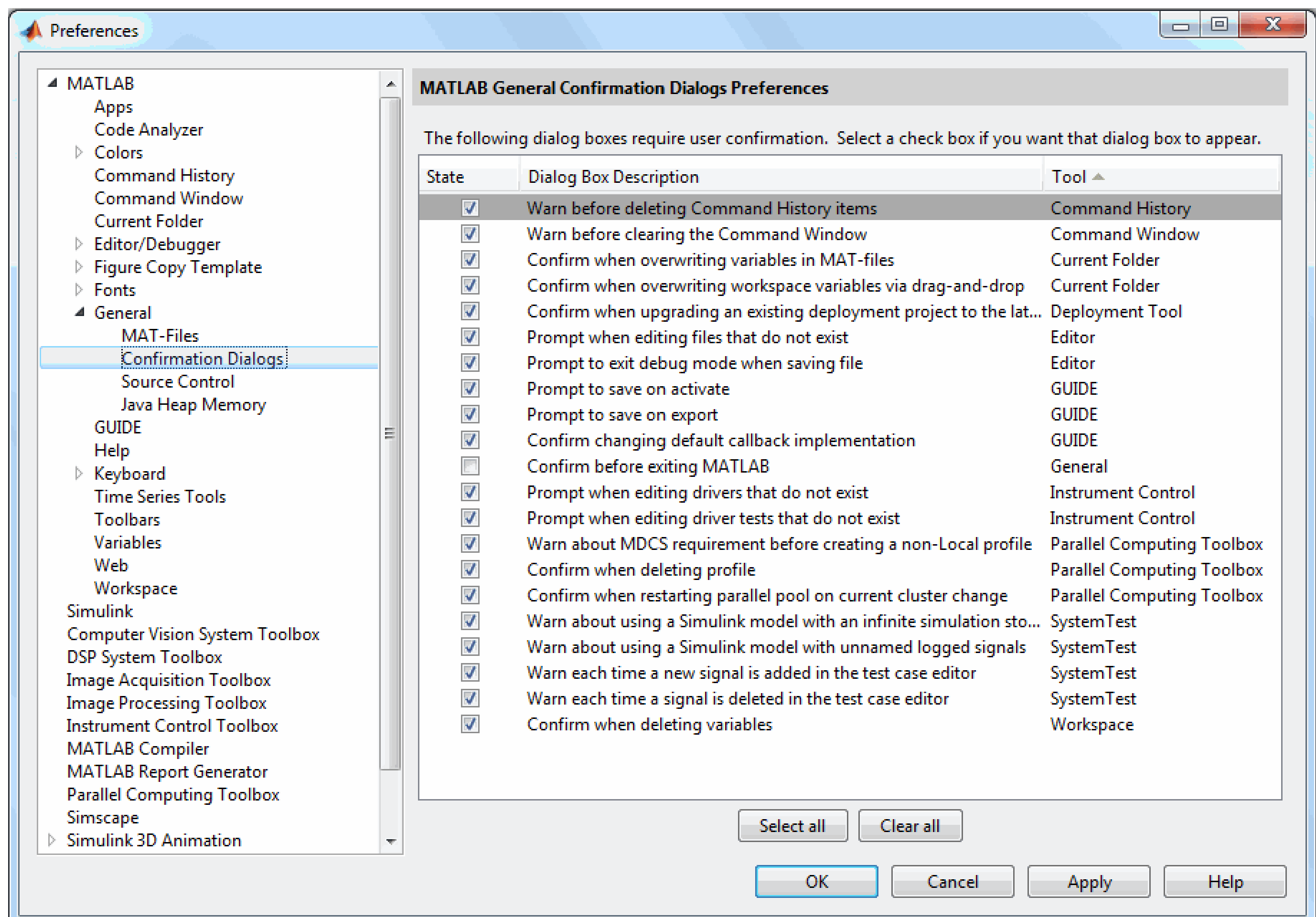
"Device Objects" on page 1-22

"IVI Configuration Store" on page 1-22

"IVI Instruments" on page 1-23

Accessing General Preferences

You access the general preferences from MATLAB – on the **Home** tab, in the **Environment** section, click **Preferences**. In the Preferences dialog box, there are two options listed for Instrument Control under the **MATLAB > General** node, in **Confirmation Dialogs**.



MATLAB Instrument Driver Editor

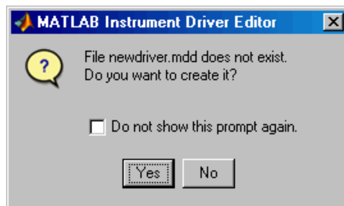
The first option for Instrument Control is related to the MATLAB Instrument Driver Editor (`midedit`).

When the option **Prompt when editing drivers that do not exist** is selected, if you open the MATLAB Instrument Driver Editor while specifying a driver file that does not exist, you get a prompt asking if you want to create a new driver file.

For example, the command

```
midedit ('newdriver')
```

generates the prompt



If you select **Do not show this prompt again**, the corresponding check box in the Preferences dialog box is cleared, in which case the MATLAB Instrument Driver Editor creates new driver files without prompting. To reactivate the prompt, select the option on the Preferences dialog box.

MATLAB Instrument Driver Testing Tool

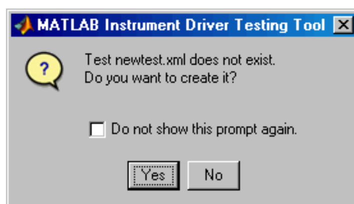
The second option for Instrument Control is related to the MATLAB Instrument Driver Testing Tool (`midtest`).

When the option **Prompt when editing driver tests that do not exist** is selected, if you open the MATLAB Instrument Driver Testing Tool while specifying a driver test file that does not exist, you get a prompt asking if you want to create a new test file.

For example, the command

```
midtest ('newtest')
```

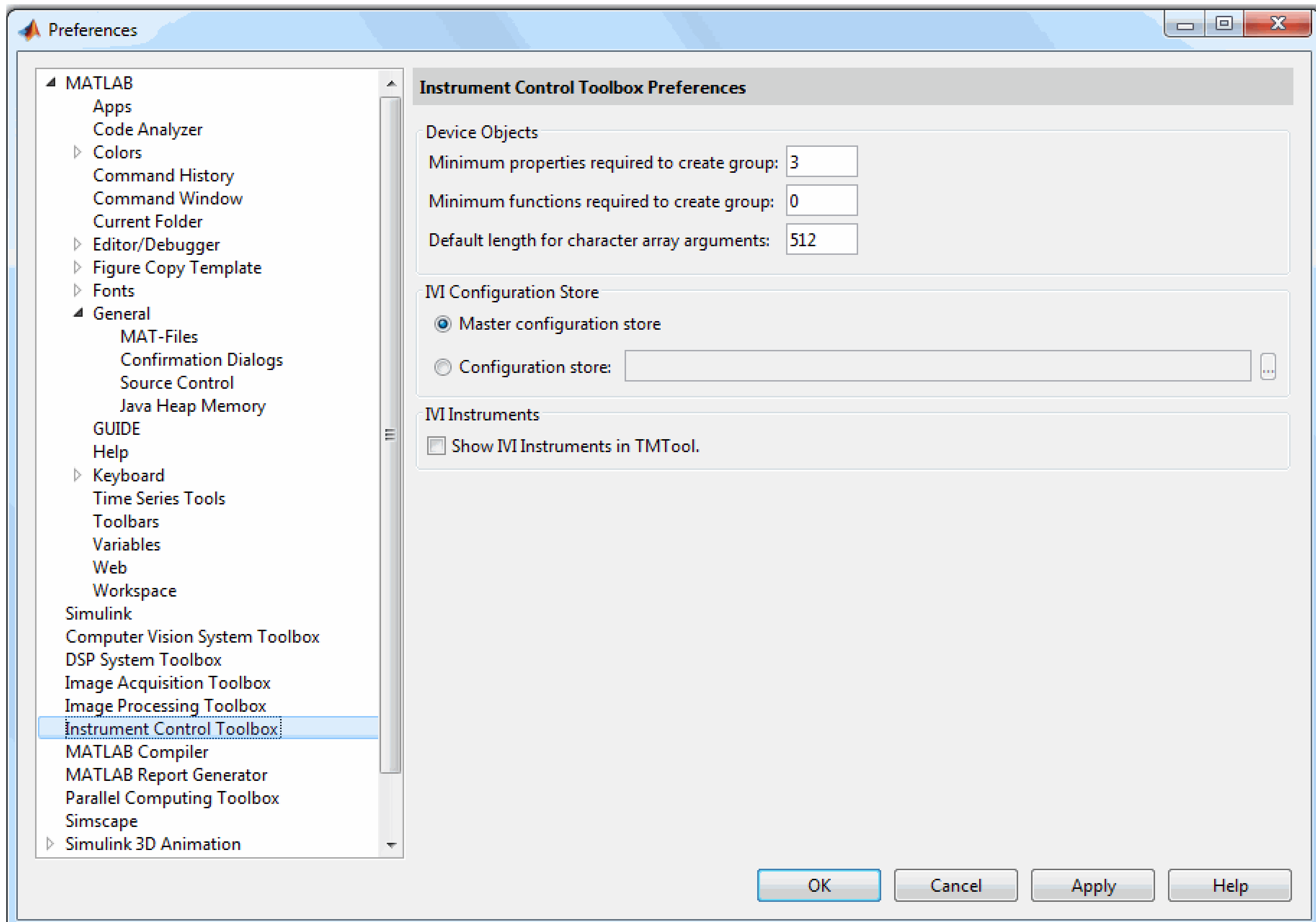
generates the prompt



If you select **Do not show this prompt again**, the corresponding check box in the Preferences dialog box is cleared, in which case the MATLAB Instrument Driver Testing Tool creates new driver test files without prompting. To reactivate the prompt, check the option on the Preferences dialog box.

Device Objects

You access other Instrument Control Preferences by selecting the **Instrument Control Toolbox** node in the tree.



The **Device Objects** section of the dialog box contains preferences related to the construction and use of device objects for VXI*plug&play* and IVI-C drivers.

Here you set the minimum number of properties and functions required to create a device object group, and the default size of character arrays passed as output arguments to device object functions.

Set the default size for these character arrays in the Preferences dialog box to ensure that they are large enough to accommodate any string returned to them by any device object functions. You can reduce the default character array size to avoid unnecessary memory usage, as long as they are still large enough to accommodate any expected strings.

IVI Configuration Store

The **IVI Configuration Store** section of the dialog box contains preferences related to the construction and use of IVI configuration store objects when you are working in the Command Window or in the **Test and Measurement Tool**.

You can select either a master configuration store or a user-defined configuration store. If you choose a user-defined configuration store, you must provide its file name.

IVI Instruments

You can use the IVI-C Wrappers functionality from the Test & Measurement Tool. View the IVI-C nodes in the Tool by selecting this **Show IVI Instruments in TMTool** preference in MATLAB.

For more information, see “IVI-C Class Compliant Wrappers in Test & Measurement Tool” on page 14-19.

Interface and Property Help

In this section...
“instrhelp Function” on page 1-24
“propinfo Function” on page 1-24
“instrsupport Function” on page 1-25
“Overview Help” on page 1-25
“Documentation Examples” on page 1-26
“Online Support” on page 1-26

instrhelp Function

You can use the `instrhelp` function to:

- Display command-line help for functions and properties.
- List all the functions and properties associated with a specific instrument object.

An instrument object is not only for you to obtain this information, but also to display all functions and properties associated with the object, as well as the constructor help. For example, to see this information for a GPIB object, type:

```
instrhelp gpib
```

To display help for the `EOIMode` property, type:

```
instrhelp EOIMode
```

You can also display help for an existing instrument object. For example, to display help for the `MemorySpace` property associated with a VISA-GPIB-VXI object, type:

```
v = visa('keysight','GPIB-VXI0::80::INSTR');
out = instrhelp(v,'MemorySpace');
```

Alternatively, you can display help via the Workspace browser by right-clicking an instrument object and selecting **Instrument Help** from the context menu.

propinfo Function

You can use the `propinfo` function to return the characteristics of the Instrument Control Toolbox properties. For example, you can find the default value for any property using this function. `propinfo` returns a structure containing the following fields:

Field Name	Description
`Type`	The property data type. Possible values are `any`, `ASCII value`, `callback`, `double`, `string`, and `struct`.
`Constraint`	The type of constraint on the property value. Possible values are `ASCII value`, `bounded`, `callback`, `enum`, and `none`.

Field Name	Description
`ConstraintValue`	The property value constraint. The constraint can be a range of values or a list of character vector values.
`DefaultValue`	The property default value.
`ReadOnly`	The condition under which a property is read only. Possible values are `always`, `never`, `whileOpen`, and `whileRecording`.
`InterfaceSpecific`	If the property is interface-specific, a `1` is returned. If the property is supported for all interfaces, a `0` is returned.

For example, to display the property characteristics for the `EOIMode` property associated with the GPIB object `g`,

```
g = gpib('ni',0,2);
EOIinfo = propinfo(g,'EOIMode')

EOIinfo =
                 Type: 'string'
           Constraint: 'enum'
      ConstraintValue: {2x1 cell}
         DefaultValue: 'on'
             ReadOnly: 'never'
    InterfaceSpecific: 1
```

This information tells you the following:

- The property value data type is a character vector.
- The property value is constrained as an enumerated list of values.
- There are two possible property values.
- The default value is `on`.
- The property can be configured at any time (it is never read-only).
- The property is not supported for all interfaces.

To display the property value constraints,

```
EOIinfo.ConstraintValue
ans =
    'on'
    'off'
```

instrsupport Function

Execute this function to get diagnostic information for all installed hardware adaptors on your system. The information is stored in a text file, `instrsupport.txt` in your current folder and you can use this information to troubleshoot issues.

Overview Help

The overview help lists the toolbox functions grouped by usage. You can display this information by typing

```
help instrument
```

For the code for any function, type

```
type function_name
```

Documentation Examples

This guide provides detailed examples that show you how to communicate with all supported interface types. These examples are contained in all the appropriate sections throughout the documentation. For example, in the sections about Bluetooth communication, you will find examples of communicating with Bluetooth instruments.

The examples use specific peripheral instruments, GPIB controllers, string commands, address information, and so on. If your instrument accepts different string commands, or if your hardware is configured to use different address information, you should modify the examples accordingly.

There are also some examples that show special applications of the Toolbox or show complete workflows of certain features or interfaces. These appear in the **Examples** list at the top of the Instrument Control Toolbox Documentation Center main page. You do not need an instrument connected to your computer to use these tutorials as they use prerecorded data.

Online Support

For online support of Instrument Control Toolbox software, visit the Web site `https://www.mathworks.com/support/`. This site includes documentation, examples, solutions, downloads, system requirements, and contact information.

Instrument Control Session

The instrument control session consists of the steps you are likely to take when communicating with your instrument. This chapter highlights some of the differences between interface objects and device objects for each of these steps, to help you decide which to use in communicating with your instrument. Whether you use interface objects or device objects, the basic steps of the instrument control session remain the same, as outlined in this chapter.

- “Creating Instrument Objects” on page 2-2
- “Connecting to the Instrument” on page 2-3
- “Configuring and Returning Properties” on page 2-4
- “Communicating with Your Instrument” on page 2-8
- “Disconnecting and Cleaning Up” on page 2-9
- “Summary” on page 2-10
- “Instrument Control Toolbox Properties” on page 2-11

Creating Instrument Objects

In this section...
“Overview” on page 2-2
“Interface Objects” on page 2-2
“Device Objects” on page 2-2

Overview

Instrument objects are the toolbox components you use to access your instrument. They provide a gateway to the functionality of your instrument and allow you to control the behavior of your application. The Instrument Control Toolbox software supports two types of instrument objects:

- Interface objects are associated with a specific interface standard such as GPIB or VISA. They allow you to communicate with any instrument connected to the interface.
- Device objects are associated with a MATLAB instrument driver. They allow you to communicate with your instrument using properties and functions defined in the driver for a specific instrument model.

Interface Objects

An interface object represents a channel of communication. For example, an interface object might represent a device at address 4 on the GPIB, even though there is nothing specific about what kind of instrument this may be.

To create an instrument object, call the constructor for the type of interface (`gpib`, `serialport`, `tcpclient`, `tcpserver`, `udpport`, or `visadev`), and provide appropriate interface information, such as address for GPIB, remote host for TCP/IP, or port number for serial.

For detailed information on interface objects and how to create and use them, see “Create Interface Object” on page 3-2.

Device Objects

A device object represents an instrument rather than an interface. As part of that representation, a device object must also be aware of the instrument driver.

You create a device object with the `icdevice` function. A device object requires a MATLAB instrument driver and some form of instrument interface, which can be an interface object, a VISA resource name, or an interface implied in an IVI configuration.

For detailed information on device objects and how to create and use them, see “Device Objects” on page 12-2.

Connecting to the Instrument

Before you can use an instrument object to write or read data, you must connect it to the instrument. You connect an interface object to the instrument with the `fopen` function; you connect a device object to the instrument with the `connect` function.

You can examine the `Status` property to verify that the instrument object is connected to the instrument.

```
obj.Status
ans =
open
```

Some properties of the object are read-only while the object is connected and must be configured before connecting. Examples of interface object properties that are read-only when the object is connected include `InputBufferSize` and `OutputBufferSize`. You can determine when a property is configurable with the `propinfo` function or by referring to the properties documentation.

Configuring and Returning Properties

In this section...

"Configuring Property Names and Property Values" on page 2-4
"Returning Property Names and Property Values" on page 2-4
"Using Tab Completion for Functions" on page 2-4
"Property Inspector" on page 2-6

Configuring Property Names and Property Values

You establish the desired instrument object behavior by configuring property values. You can configure property values using the `set` function or the dot notation, or by specifying property name/property value pairs during object creation. You can return property values using the `get` function or the dot notation.

Interface objects possess two types of properties: *base properties* and *interface-specific properties*. (These properties pertain only to the interface object itself and to the interface, *not* to the instrument.) Base properties are supported for all interface objects (serial port, GPIB, VISA-VXI, and so on), while interface-specific properties are supported only for objects of a given interface type. For example, the `BaudRate` property is supported only for serial port and VISA-serial objects.

Device objects also possess two types of properties: *base properties* and *device-specific properties*. While device objects possess base properties pertaining to the object and interface, they also possess any number of device-specific properties as defined in the instrument driver for configuring the instrument. For example, a device object representing an oscilloscope might posses such properties as `DisplayContrast`, `InputRange`, and `MeasurementMode`. When you set these properties you are directly configuring the oscilloscope settings.

Returning Property Names and Property Values

Once the instrument object is created, you can use the `set` function to return all its configurable properties to a variable or to the command line. Additionally, if a property has a finite set of character vector values, `set` returns these values.

Using Tab Completion for Functions

To get a list of options you can use on the function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For example, when you create a `gpib` object, you can get a list of installed vendors:

```
g = gpib('
```

When you press **Tab** after the parentheses and single quote, as shown here, the list of installed GPIB vendors displays, such as `keysight`, `ics`, `mcc`, and `ni`.

The format for the GPIB object constructor function is:

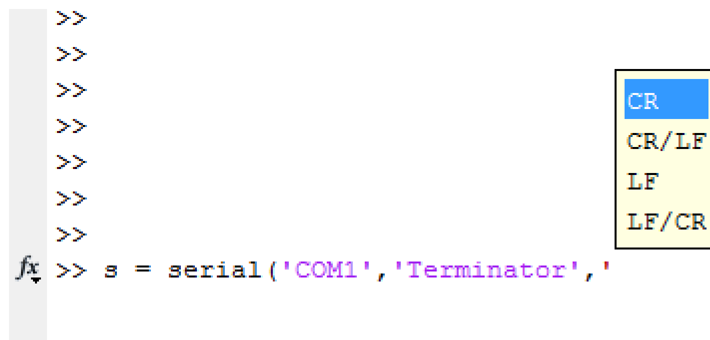
```
g = gpib('vendor',boardindex,primaryaddress)
```

When you press **Tab** where a field should appear, you get the list of options for that field. The other interface objects, such as Bluetooth, Serial, TCP/IP, etc., also include this capability on their object constructor functions.

You can also get the values for property-value pairs. For example, to get the possible terminator values when creating a serial object, type:

```
s = serial('COM1','Terminator','
```

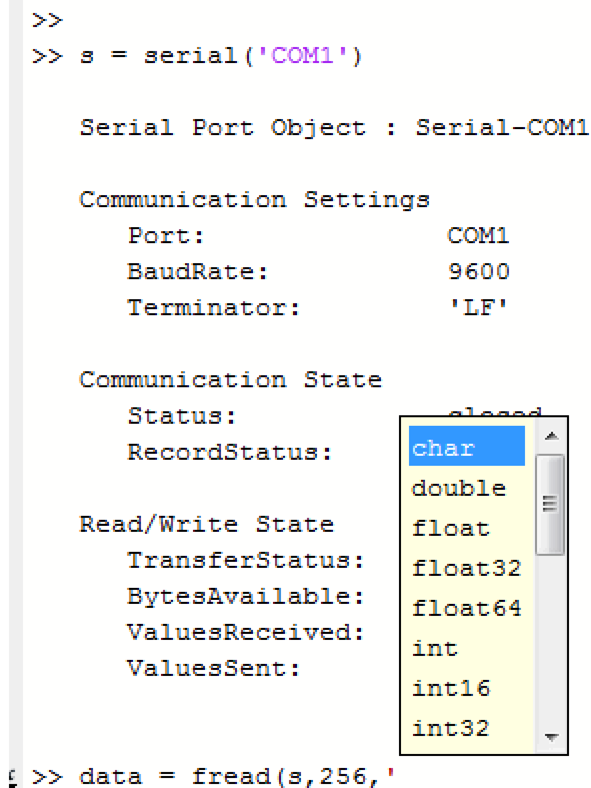
Press **Tab** after typing the single quote after `Terminator` to get the possible values for that property, as shown here.



Many of the other toolbox functions also have tab completion. For example, when using the `fread` function you can specify the precision type using tab completion.

```
data = fread(s,256,'
```

Press **Tab** after typing the single quote after the `size` (256 values in this example), since precision is the next argument the `fread` function takes, to get the possible values for the precision types, such as `'double'`, `'int16'`, etc.



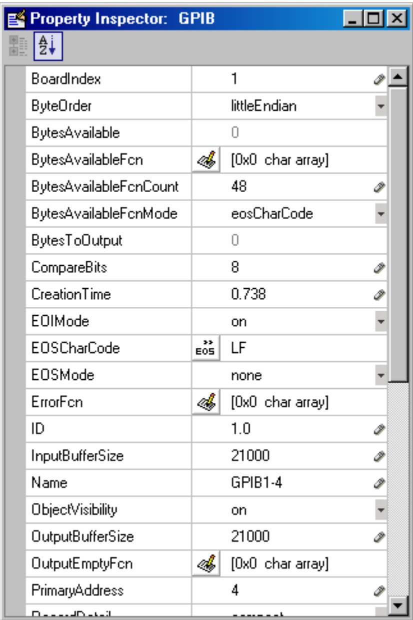
When the list of possible values is long, a scroll bar appears in the pop-up window, as shown in this example.

Property Inspector

The Property Inspector enables you to inspect and set properties for one or more instrument objects. It provides a list of all properties and displays their current values.

Settable properties in the list are associated with an editing device that is appropriate for the values accepted by the particular property. For example, a callback configuration GUI to set `ErrorFcn`, a pop-up menu to set `RecordMode`, and a text field to specify the `TimerPeriod`. The values for read-only properties are grayed out.

You open the Property Inspector with the `inspect` function. Alternatively, you can open the Property Inspector via the Workspace browser by right-clicking an instrument object and selecting **Call Property Inspector** from the context menu, or by double-clicking the object.


Property Inspector: GPIB
BoardIndex 1
ByteOrder littleEndian
BytesAvailable 0
BytesAvailableFcn [0x0 char array]
BytesAvailableFcnCount 48
BytesAvailableFcnMode eosCharCode
BytesToOutput 0
CompareBits 8
CreationTime 0.738
EOIMode on
EOSCharCode LF
EOSMode none
ErrorFcn [0x0 char array]
ID 1.0
InputBufferSize 21000
Name GPIB1-4
ObjectVisibility on
OutputBufferSize 21000
OutputEmptyFcn [0x0 char array]
PrimaryAddress 4

Communicating with Your Instrument

In this section...

"Interface Objects and Instrument Commands" on page 2-8
"Device Objects and Instrument Drivers" on page 2-8

Interface Objects and Instrument Commands

Communicating with your instrument involves sending and receiving commands, settings, responses, and data. The level of communication depends on the type of instrument object you use.

To communicate through the interface object, you need to send instrument commands, and you receive information as the instrument sends it. Therefore, you have to know the syntax specific to the instrument itself. For example, if the instrument requires the command `'*RST'` to initiate its action, then that is exactly the command that must be sent to the interface object.

Text commands and binary data are sent directly to the instrument and received from the instrument with such functions as `fprintf`, `fwrite`, `fgets`, `fread`, and others.

Device Objects and Instrument Drivers

To communicate through a device object, you access object properties with the `set` and `get` commands, and you execute driver functions with the `invoke` command. The `invoke` command for a device object employs methods and arguments defined by the instrument driver. So using device objects does not require you to use instrument-specific commands and syntax.

For information on creating, editing, and importing instrument drivers, see "MATLAB Instrument Driver Editor Overview" on page 19-2.

Disconnecting and Cleaning Up

In this section...

"Disconnecting an Instrument Object" on page 2-9
"Cleaning Up the MATLAB Workspace" on page 2-9

Disconnecting an Instrument Object

When you no longer need to communicate with the instrument, you should disconnect the object. Interface objects are disconnected with the `fclose` function; device objects are disconnected with the `disconnect` function.

You can examine the `Status` property to verify that the object is disconnected from the instrument.

```
obj.Status
ans =
closed
```

Cleaning Up the MATLAB Workspace

When you no longer need the instrument object, you should remove it from memory with the `delete` function.

```
delete(obj)
```

A deleted instrument object is *invalid*, which means that you cannot connect it to the instrument. In this case, you should remove the object from the MATLAB workspace. To remove instrument objects and other variables from the MATLAB workspace, use the `clear` command.

```
clear obj
```

If you use `clear` on an object that is connected to an instrument, the object is removed from the workspace but remains connected to the instrument. You can restore cleared instrument objects to the MATLAB workspace with the `instrfind` function.

Summary

In this section...
“Advantages of Using Device Objects” on page 2-10
“When to Use Interface Objects” on page 2-10

Advantages of Using Device Objects

Should you use interface objects or device objects to communicate with your instrument? Generally, device objects make instrument control easier and they offer greater flexibility to the user compared to using interface objects.

Because of the advantages offered by using device objects for communicating with your instrument, you should use device objects whenever possible. Some of these advantages are

- You do not need to know instrument-specific commands
- You can use standard VXI*plug&play* or IVI instrument drivers provided by your instrument vendor or other party
- You can use a MATLAB instrument driver to control your instrument. To get a MATLAB instrument driver, you can
  - Convert a VXI*plug&play* or IVI driver
  - Use a MATLAB driver that is shipped with the toolbox
  - Create it yourself or modify a similar driver
  - Install it from a third party, such as MATLAB Central

  You can create, convert, or customize a MATLAB instrument driver with the MATLAB Instrument Driver Editor tool (`midedit`).

When to Use Interface Objects

In some circumstances, using device objects to communicate with your instrument would be impossible or impractical. You might need to use interface objects if

- Your instrument does not have a standard instrument driver supported by the Instrument Control Toolbox software.
- You are using a streaming application (typically serial, UDP, or TCP/IP interface) to notify you of some occurrence.
- Your application requires frequent changes to communication channel settings.

Instrument Control Toolbox Properties

The following properties are available in the toolbox.

- ActualLocation
- Alias
- BaudRate
- BoardIndex
- BreakInterruptFcn
- BusManagementStatus
- ByteOrder
- BytesAvailableFcn
- BytesAvailableFcnCount
- BytesAvailableFcnMode
- BytesToOutput
- ChassisIndex
- CompareBits
- ConfirmationFcn
- DataBits
- DatagramAddress
- DatagramPort
- DatagramReceivedFcn
- DatagramTerminateMode
- DataTerminalReady
- DriverName
- DriverSessions
- DriverType
- EOIMode
- EOSCharCode
- EOSMode
- ErrorFcn
- FlowControl
- HandshakeStatus
- HardwareAssets
- HwIndex
- HwName
- InputBufferSize
- InputDatagramPacketSize
- InstrumentModel
- Interface

- InterfaceIndex
- InterruptFcn
- LANName
- LocalHost
- LocalPort
- LocalPortMode
- LogicalAddress
- LogicalName
- LogicalNames
- ManufacturerID
- MappedMemoryBase
- MappedMemorySize
- MasterLocation
- MemoryBase
- MemoryIncrement
- MemorySize
- MemorySpace
- ModelCode
- Name
- NetworkRole
- ObjectVisibility
- OutputBufferSize
- OutputDatagramPacketSize
- OutputEmptyFcn
- Parent
- Parity
- PinStatus
- PinStatusFcn
- Port
- PrimaryAddress
- ProcessLocation
- PublishedAPIs
- ReadAsyncMode
- RecordDetail
- RecordMode
- RecordName
- RecordStatus
- RemoteHost
- RemotePort

- RequestToSend
- Revision
- RsrcName
- SecondaryAddress
- SerialNumber
- ServerDescription
- Sessions
- Slot
- SoftwareModules
- SpecificationVersion
- Status
- StopBits
- Tag
- Terminator
- Timeout
- TimerFcn
- TimerPeriod
- TransferDelay
- TransferStatus
- TriggerFcn
- TriggerLine
- TriggerType
- Type
- UserData
- ValuesReceived
- ValuesSent
- Vendor

Using Interface Objects

The instrument control session using interface objects consists of all the steps described in the following sections.

- "Create Interface Object" on page 3-2
- "Configure and Return Properties" on page 3-3
- "Write and Read Data" on page 3-5
- "Use SCPI Commands" on page 3-9

Create Interface Object

Connect to Instrument

An interface object represents a connection to an instrument over one of the supported interfaces. The following instrument communication interfaces are supported.

- "Serial Port Interface"
- "TCP/IP Interface"
- "UDP Interface"
- "VISA Interface"

Create a connection to an instrument by calling the following object creation functions.

Interface Object Creation Functions

Constructor	Description
`serialport`	Create a serial port object.
`tcpclient`, `tcpserver`	Create a TCP/IP client or server object.
`udpport`	Create a UDP object.
`visadev`	Create a VISA-TCP/IP, VISA-Socket, VISA-USB, VISA-GPIB, VISA-Serial, VISA-VXI, or VISA-PXI object.

Configure Properties

Instrument objects contain properties that reflect the functionality of your instrument. You control the behavior of your instrument control application by configuring values for these properties.

As described in "Configure and Return Properties" on page 3-3, you can configure certain properties during object creation using name-value pairs. You can also set properties after object creation using dot notation. To configure callback properties, you must use the `configureCallback` and `configureTerminator` functions.

See Also
`serialport` | `tcpclient` | `tcpserver` | `udpport` | `visadev`

Related Examples
- "Configure and Return Properties" on page 3-3
- "Write and Read Data" on page 3-5

Configure and Return Properties

Establish the desired instrument object behavior by configuring property values. You can configure property values using dot notation or by specifying name-value arguments during object creation. You can return property values using dot notation.

Set Property Values During Object Creation and View Properties

You can set certain property values using name-value arguments during instrument object creation. For example, set the `serialport` object property `Timeout` during object creation.

```
s = serialport("COM4",9600,Timeout=20)
```

```
s = 

  Serialport with properties:

                 Port: "COM4"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

You can set multiple property values during object creation using the name-value argument syntax.

Once the instrument object is created, you can return properties. For example, the properties for the `serialport` object `s` are shown as follows. Click `Show all properties` to view all properties of `s` and their values.

```
s = 

  Serialport with properties:

                 Port: "COM4"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, all methods

                      Port: "COM4"
                  BaudRate: 9600
         NumBytesAvailable: 0

                 ByteOrder: "little-endian"
                  DataBits: 8
                  StopBits: 1
                    Parity: "none"
               FlowControl: "none"
                   Timeout: 20
                Terminator: "LF"

     BytesAvailableFcnMode: "off"
    BytesAvailableFcnCount: 64
         BytesAvailableFcn: []
           NumBytesWritten: 0
```

```
         ErrorOccurredFcn: []
                 UserData: []
```

You can find a full list of properties, which ones you can configure, and how to configure them in the respective interface object documentation:

- `serialport` Properties on page 24-324
- `tcpclient` Properties on page 24-348
- `tcpserver` Properties on page 24-360
- `udpport` Properties on page 24-377
- `visadev` Properties on page 24-394

Set Property Values

To display the current value for one property, use dot notation with the property name.

```
s.Timeout
```

```
ans =

    20
```

Configure property values using dot notation.

```
device.ByteOrder = "big-endian"
```

You can also configure certain properties during object creation using optional name-value arguments.

```
device = serialport("COM4",9600,DataBits=5);
```

Use `configureTerminator` to set the `Terminator` property. This property cannot be set using name-value arguments or dot notation.

```
configureTerminator(device,"CR/LF")
```

Callback properties must be configured using the `configureCallback` function. Use `configureCallback` to set the `BytesAvailableFcnMode`, `BytesAvailableFcn`, and `BytesAvailableFcnCount` properties. These properties cannot be set using name-value arguments or dot notation.

```
configureCallback(device,"terminator",@callbackFcn)
configureCallback(device,"byte",50,@callbackFcn)
```

See Also

`serialport` | `tcpclient` | `tcpserver` | `udpport` | `visadev`

Related Examples

- “Create Interface Object” on page 3-2
- “Write and Read Data” on page 3-5

Write and Read Data

Before Performing Write or Read Operations

Communicating with your instrument involves writing and reading data. For example, you might write a text command to a function generator that queries its peak-to-peak voltage, and then read back the voltage value as a double-precision array.

Before performing a write or read operation, consider the following:

- The Instrument Control Toolbox automatically manages the data transferred between the MATLAB workspace and the instrument. For many common applications, you can ignore the buffering and data flow process. However, if you are transferring a large number of values or debugging your application, you might need to be aware of how this process works.
- For many instruments, writing text data means writing string commands that change instrument settings, prepare the instrument to return data or status information, and so on. You can use the `writeline` function to write text data and the write terminator value is automatically appended to the data being written.
- Writing binary data means writing numerical values to the instrument such as calibration or waveform data. You can use the `write` function to write numeric or text data.
- You can also write binary data as a block of values. Use the `writebinblock` function to write a binblock of data.
- Read operations in the Instrument Control Toolbox are synchronous. A synchronous operation blocks access to the command line until the read operation completes execution.
  - When you call `read`, the function suspends MATLAB execution until the specified number of values is read or a timeout occurs.
  - When you call `readline`, the function suspends MATLAB execution until a terminator is read or a timeout occurs.
  - When you call `readbinblock`, the function suspends MATLAB execution until the number of values specified in the binblock is read or a timeout occurs.

Writing Data

Functions Associated with Writing Data

Function Name	Description
`write`	Write binary data to the instrument
`writeline`	Write a line of ASCII data to the instrument
`writebinblock`	Write one binblock of data to the instrument
`writeread`	Write command and read response

Note The `writebinblock` function is not available for the `udpport` interface. The `writeread` function is not available for the `tcpserver` and `udpport` interfaces.

Properties Associated with Writing Data

Property Name	Description
Timeout	Allowed time to complete write and read operations
Terminator	Terminator character for writing and reading text data
NumBytesWritten	Total number of bytes written

You can modify the value of `Timeout` using dot notation and `Terminator` using the `configureTerminator` function. `NumBytesWritten` is a read-only property.

Writing Text Data Versus Writing Binary Data

For many instruments, writing text data means writing string commands that change instrument settings, prepare the instrument to return data or status information, and so on. Writing binary data means writing numerical values to the instrument such as calibration or waveform data.

You can write text data with the `writeline` function. The `writeline` function formats the data as a string and automatically appends the terminator. You can write binary data with the `write` function. The `write` function writes data as `uint8` by default, but you can specify other data types using a name-value argument.

The following example illustrates writing text data and binary data to a Tektronix TDS 210 oscilloscope. The text data consists of string commands, while the binary data is a waveform that is to be downloaded to the scope and stored in its memory:

1. **Create an instrument object** — Create the VISA-GPIB object `g` associated with a National Instruments GPIB controller with board index 0 and an instrument with primary address 1.

```
g = visadev("GPIB0::1::0::INSTR");
```

2. **Write data** — Write string commands using `writeline` to configure the scope to store binary waveform data in memory location A.

```
writeline(g,"DATA:DESTINATION REFA");
writeline(g,"DATA:ENCDG SRPbinary");
writeline(g,"DATA:WIDTH 1");
writeline(g,"DATA:START 1");
```

Create the waveform data.

```
t = linspace(0,25,2500);
data = round(sin(t)*90 + 127);
```

Write the binary waveform data to the scope using `write`.

```
cmd = double('CURVE #42500');
write(g,[cmd data]);
```

The `NumBytesWritten` property indicates the total number of bytes that were written to the instrument.

```
g.NumBytesWritten
```

```
ans =
```

2581

3 **Disconnect and clean up** — Use `clear` to disconnect the instrument from the VISA-GPIB object `g` and to clear it from the MATLAB workspace when you are done working with it.

```
clear g
```

Reading Data

Functions Associated with Reading Data

Function Name	Description
`read`	Read binary data from the instrument
`readline`	Read a line of ASCII data from the instrument
`readbinblock`	Read one binblock of data from the instrument
`writeread`	Write ASCII command and read response

Note The `readbinblock` function is not available for the `udpport` interface. The `writeread` function is not available for the `tcpserver` and `udpport` interfaces.

Properties Associated with Reading Data

Property Name	Description
`Timeout`	Allowed time to complete write and read operations
`Terminator`	Terminator character for writing and reading text data
`NumBytesAvailable`	Number of bytes available to read

You can modify the value of `Timeout` using dot notation and `Terminator` using the `configureTerminator` function. `NumBytesAvailable` is a read-only property.

Reading Text Data Versus Reading Binary Data

For many instruments, reading text data means reading string data that reflect instrument settings, status information, and so on. Reading binary data means reading numerical values from the instrument.

You can read text data with the `readline` function. The `readline` function reads data until the first occurrence of the terminator and returns it as a string without the terminator. You can read binary data with the `read` function. The `read` function returns a specified number of values as `uint8` data by default, but you can specify other data types using a name-value argument.

The following example illustrates reading text data and binary data from a Tektronix TDS 210 oscilloscope, which is displaying a periodic input signal with a nominal frequency of 1.0 kHz.

1 **Create an instrument object** — Create the VISA-GPIB object `g` associated with a National Instruments GPIB controller with board index 0 and an instrument with primary address 1.

```
g = visadev("GPIB0::1::0::INSTR");
```

2 **Write and read data** — Write the `*IDN?` command to the instrument using `writeline`, and then read back the result of the command using `readline`.

```
writeline(g,"*IDN?")
g.NumBytesAvailable

ans =

    56

idn = readline(g)

idn =

    "TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04"
```

You can also use the `writeread` function to perform the same operation. Write the command to your instrument and read the response.

```
idn = writeread(g,"*IDN?")

idn =

    "TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04"
```

Note The `writeread` function is not available for the `tcpserver` and `udpport` interfaces.

Configure the scope to return the period of the input signal. Use `read` to read the period as a string.

```
writeline(g,"MEASUREMENT:MEAS1:TYPE PERIOD")
writeline(g,"MEASUREMENT:MEAS1:VALUE?")
period = read(g,9,"string")


period =

    "1.006E-3
     "
```

3 **Disconnect and clean up** — Use `clear` to disconnect the instrument from the VISA-GPIB object `g` and to clear it from the MATLAB workspace when you are done working with it.

```
clear g
```

See Also

`serialport` | `tcpclient` | `tcpserver` | `udpport` | `visadev`

Related Examples

- “Create Interface Object” on page 3-2
- “Configure and Return Properties” on page 3-3
- “Use SCPI Commands” on page 3-9

Use SCPI Commands

Standard Commands for Programmable Instruments, or SCPI commands, are an ASCII-based set of pre-defined commands and responses. They use the same data format across all SCPI compliant instruments. You can use SCPI commands with the Instrument Control Toolbox and the MATLAB programming environment to control multiple instruments using similar functions. You can access a common functionality in instruments without changing your programming environment. SCPI commands are simple and flexible and accept a range of parameter formats. This allows you to easily program your instrument. The response to SCPI commands can be status information or data. You can define the format of the data independently of the device or the measurement. For more information refer to the IVI Foundation SCPI Specifications.

Use the `writeline` function on instrument objects to send SCPI commands. Then, use the `readline` function to read the response. You can also use the `writeread` function to send SCPI commands that require a response.

Commonly Used SCPI Commands

Commands	Functionality
*CLS	Clear the status
*ESE	Enable standard event
*ESE?	Query if event is enabled and standard
*ESR?	Query standard event status register
*IDN?	Query instrument identification
*OPC	Operation complete
*OPC?	Query if operation is complete
*RST	Instrument reset
*SRE	Enable service request
*SRE?	Query id service request is enabled
*STB?	Query read of status byte
*TST?	Query instrument self test
*WAI	Wait to continue

See Also

`serialport` | `tcpclient` | `tcpserver` | `udpport` | `visadev`

Related Examples

- “Create Interface Object” on page 3-2

Controlling Instruments Using GPIB

This chapter describes specific issues related to controlling instruments that use the GPIB interface.

- “GPIB Overview” on page 4-2
- “Write and Read GPIB Data” on page 4-10
- “Transition Your Code to VISA-GPIB Interface” on page 4-16

GPIB Overview

In this section...

"What Is GPIB?" on page 4-2
"Important GPIB Features" on page 4-2
"GPIB Lines" on page 4-3
"Status and Event Reporting" on page 4-6

What Is GPIB?

GPIB is a standardized interface that allows you to connect and control multiple devices from various vendors. GPIB is also referred to by its original name HP-IB, or by its IEEE designation IEEE-488. GPIB functionality has evolved over time, and is described in several specifications:

- The IEEE 488.1-1975 specification defines the electrical and mechanical characteristics of the interface and its basic functional characteristics.
- The IEEE-488.2-1987 specification builds on the IEEE 488.1 specification to define an acceptable minimum configuration and a basic set of instrument commands and common data formats.
- The Standard Commands for Programmable Instrumentation (SCPI) specification builds on the commands given by the IEEE 488.2 specification to define a standard instrument command set that can be used by GPIB or other interfaces.

For many GPIB applications, you can communicate with your instrument without detailed knowledge of how GPIB works. Communication is established through a VISA-GPIB object, which you create using `visadev`.

If your application is straightforward, or if you are already familiar with the GPIB specifications, you can begin with "Get Started with GPIB Interface" on page 5-11. If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to "Creating Instrument Objects" on page 2-2.

Some of the GPIB functionality is required for all GPIB devices, while other GPIB functionality is optional. Additionally, many devices support only a subset of the SCPI command set, or use a different vendor-specific command set. Refer to your device documentation for a complete list of its GPIB capabilities and its command set.

Important GPIB Features

The important GPIB features are described in the following sections. For detailed information about GPIB functionality, see the appropriate references in the Appendix A.

Bus and Connector

The GPIB bus is a cable with two 24-pin connectors that allow you to connect multiple devices to each other. The bus and connector have these features and limitations:

- You can connect up to 15 devices to a bus.
- You can connect devices in a star configuration, a linear configuration, or a combination of configurations.

- To achieve maximum data transfer rates, the cable length should not exceed 20 meters total or an average of 2 meters per device. You can eliminate these restrictions by using a bus extender.

GPIB Devices

Each GPIB device must be some combination of a *Talker*, a *Listener*, or a *Controller*. A Controller is typically a board that you install in your computer. Talkers and Listeners are typically instruments such as oscilloscopes, function generators, multimeters, and so on. Most modern instruments are both Talkers and Listeners.

- Talkers — A Talker transmits data over the interface when addressed to talk by the Controller. A GPIB system can contain only one Talker at a time.
- Listeners — A Listener receives data over the interface when addressed to listen by the Controller. A GPIB system can contain up to 14 Listeners at a given time. Typically, the Controller is a Talker while one or more instruments on the GPIB are Listeners.
- Controllers — The Controller specifies which devices are Talkers or Listeners. A GPIB system can contain multiple Controllers. One of them is designated the System Controller. However, only one Controller can be active at a given time. The current active controller is the Controller-In-Charge (CIC). The CIC can pass control to an idle Controller, but only the System Controller can make itself the CIC.

  When the Controller is not sending messages, then a Talker can send messages. Typically, the CIC is a Listener while another device is enabled as a Talker.

Each Controller is identified by a unique board index number. Each Talker/Listener is identified by a unique primary address ranging from 0 to 30, and by an optional secondary address, which can be 0 or can range from 96 to 126.

GPIB Data

Two types of data can be transferred over GPIB, *instrument data* and *interface messages*:

- Instrument data — Instrument data consists of vendor-specific commands that configure your instrument, return measurement results, and so on. For a complete list of commands supported by your instrument, refer to its documentation.
- Interface messages — Interface messages are defined by the GPIB standard and consist of commands that clear the GPIB bus, address devices, return self-test results, and so on.

Data transfer consists of one byte (8 bits) sent in parallel. The data transfer rate across the interface is limited to 1 megabyte per second. However, this data rate is usually not achieved in practice, and is limited by the slowest device on the bus.

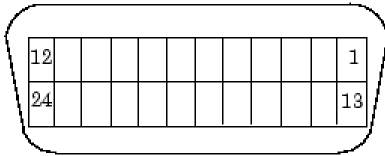
GPIB Lines

GPIB consists of 24 lines, which are shared by all instruments connected to the bus. 16 lines are used for signals, while eight lines are for ground. The signal lines are divided into these groups:

- Eight data lines
- Five interface management lines
- Three handshake lines

The signal lines use a low-true (negative) logic convention with TTL levels. This convention means that a line is low (true or asserted) when it is a TTL low level, and a line is high (false or unasserted)

when it is a TTL high level. The pin assignment scheme for a GPIB connector is shown in the following figure and table.



GPIB Pin and Signal Assignments

Pin	Label	Signal Name	Pin	Label	Signal Name
1	DIO1	Data transfer	13	DIO5	Data transfer
2	DIO2	Data transfer	14	DIO6	Data transfer
3	DIO3	Data transfer	15	DIO7	Data transfer
4	DIO4	Data transfer	16	DIO8	Data transfer
5	EOI	End Or Identify	17	REN	Remote Enable
6	DAV	Data Valid	18	GND	DAV ground
7	NRFD	Not Ready For Data	19	GND	NRFD ground
8	NDAC	Not Data Accepted	20	GND	NDAC ground
9	IFC	Interface Clear	21	GND	IFC ground
10	SRQ	Service Request	22	GND	SRQ ground
11	ATN	Attention	23	GND	ATN ground
12	Shield	Chassis ground	24	GND	Signal ground

Data Lines

The eight data lines, DIO1 through DIO8, are used for transferring data one byte at a time. DIO1 is the least significant bit, while DIO8 is the most significant bit. The transferred data can be an instrument command or a GPIB interface command.

Data formats are vendor-specific and can be text-based (ASCII) or binary. GPIB interface commands are defined by the IEEE 488 standard.

Interface Management Lines

The interface management lines control the flow of data across the GPIB interface.

GPIB Interface Management Lines

Line	Description
ATN	Used by the Controller to inform all devices on the GPIB that bytes are being sent. If the ATN line is high, the bytes are interpreted as an instrument command. If the ATN line is low, the bytes are interpreted as an interface message.
IFC	Used by the Controller to initialize the bus. If the IFC line is low, the Talker and Listeners are unaddressed, and the System Controller becomes the Controller-In-Charge.
REN	Used by the Controller to place instruments in remote or local program mode. If REN is low, all Listeners are placed in remote mode, and you cannot change their settings from the front panel. If REN is high, all Listeners are placed in local mode.
SRQ	Used by Talkers to asynchronously request service from the Controller. If SRQ is low, then one or more Talkers require service (for example, an error such as invalid command was received). You issue a serial poll to determine which Talker requested service. The poll automatically sets the SRQ line high.
EOI	If the ATN line is high, the EOI line is used by Talkers to identify the end of a byte stream such as an instrument command. If the ATN line is low, the EOI line is used by the Controller to perform a parallel poll (not supported by the toolbox).

Handshake Lines

The three handshake lines, DAV, NRFD, and NDAC, are used to transfer bytes over the data lines from the Talker to one or more addressed Listeners.

Before data is transferred, all three lines must be in the proper state. The active Talker controls the DAV line and the Listener(s) control the NRFD and NDAC lines. The handshake process allows for error-free data transmission.

Handshake Lines

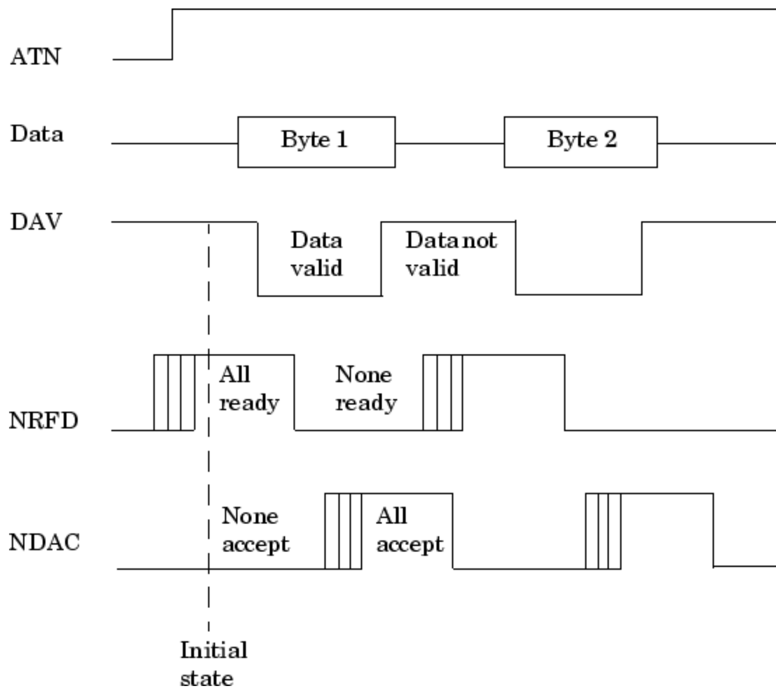
Line	Description
DAV	Used by the Talker to indicate that a byte can be read by the Listeners.
NRFD	Indicates whether the Listener is ready to receive the byte.
NDAC	Indicates whether the Listener has accepted the byte.

The handshaking process follows these steps:

1. Initially, the Talker holds the DAV line high indicating no data is available, while the Listeners hold the NRFD line high and the NDAC line low indicating they are ready for data and no data is accepted, respectively.
2. When the Talker puts data on the bus, it sets the DAV line low, which indicates that the data is valid.
3. The Listeners set the NRFD line low, which indicates that they are not ready to accept new data.
4. The Listeners set the NDAC line high, which indicates that the data is accepted.
5. When all Listeners indicate that they have accepted the data, the Talker sets the DAV line high indicating that the data is no longer valid. The next byte of data can now be transmitted.
6. The Listeners hold the NRFD line high indicating they are ready to receive data again, and the NDAC line is held low indicating no data is accepted.

Note If the ATN line is high during the handshaking process, the information is considered data such as an instrument command. If the ATN line is low, the information is considered a GPIB interface message.

The handshaking steps are shown in the following figure.

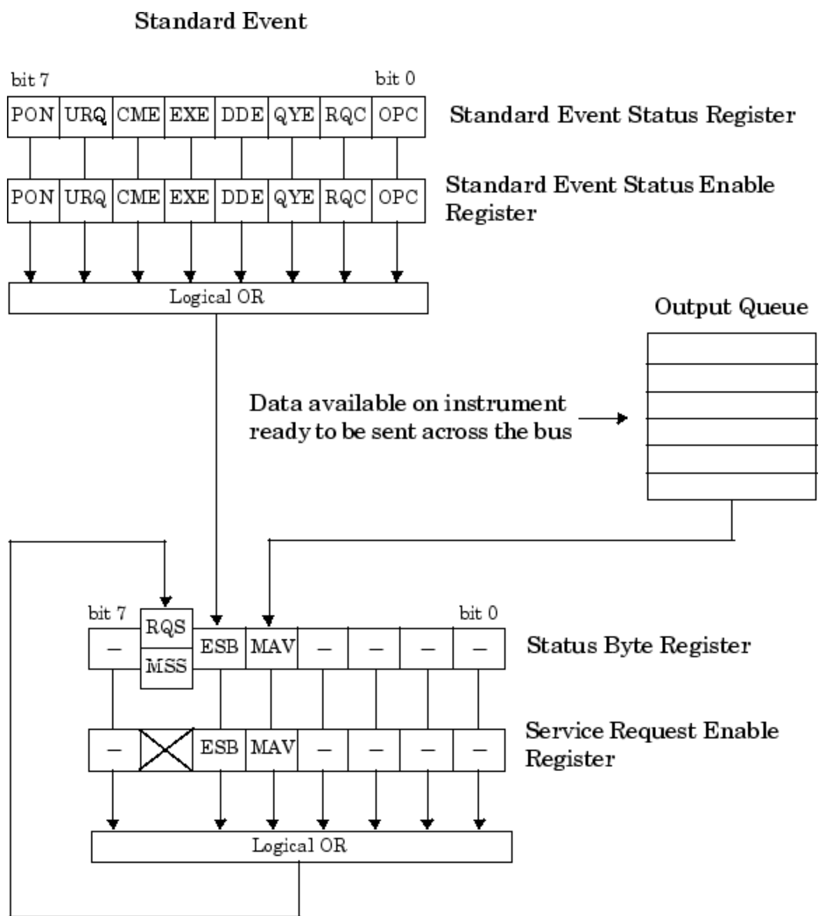


Status and Event Reporting

GPIB provides a system for reporting status and event information. With this system, you can find out if your instrument has data to return, whether a command error occurred, and so on. For many instruments, the reporting system consists of four 8-bit registers and two queues (output and event). The four registers are grouped into these two functional categories:

- Status Registers — The Status Byte Register (SBR) and Standard Event Status Register (SESR) contain information about the state of the instrument.
- Enable Registers — The Event Status Enable Register (ESER) and the Service Request Enable Register (SRER) determine which types of events are reported to the status registers and the event queue. ESER enables SESR, while SRER enables SBR.

The status registers, enable registers, and output queue are shown in the following figure.



Status Byte Register

Each bit in the Status Byte Register (SBR) is associated with a specific type of event. When an event occurs, the instrument sets the appropriate bit to 1. You can enable or disable the SBR bits with the Service Request Enable Register (SRER). You can determine which events occurred by reading the enabled SBR bits.

Status Byte Register Bits

Bit	Label	Description
0-3	–	Instrument-specific summary messages.
4	MAV	The Message Available bit indicates if data is available in the Output Queue. MAV is 1 if the Output Queue contains data. MAV is 0 if the Output Queue is empty.
5	ESB	The Event Status bit indicates if one or more enabled events have occurred. ESB is 1 if an enabled event occurs. ESB is 0 if no enabled events occur. You enable events with the Standard Event Status Enable Register.
6	MSS	The Master Summary Status summarizes the ESB and MAV bits. MSS is 1 if either MAV or ESB is 1. MSS is 0 if both MAV and ESB are 0. This bit is obtained from the `*STB?` command.
	RQS	The Request Service bit indicates that the instrument requests service from the GPIB controller. This bit is obtained from a serial poll.
7	–	Instrument-specific summary message.

For example, if you want to know when a specific type of instrument error occurs, enable bit 5 of the SRER. Additionally, enable the appropriate bit of the Standard Event Status Enable Register so that the error event of interest is reported by the ESB bit of the SBR.

Standard Event Status Register

Each bit in the Standard Event Status Register (SESR) is associated with a specific state of the instrument. When the state changes, the instrument sets the appropriate bits to 1. You can enable or disable the SESR bits with the Standard Event Status Enable Register (ESER). You can determine the state of the instrument by reading the enabled SESR bits. The SESR bits have the following descriptions.

SESR Bits

Bit	Label	Description
0	OPC	The Operation Complete bit indicates that all commands have completed.
1	RQC	The Request Control bit is not used by most instruments.
2	QYE	The Query Error bit indicates that the instrument attempted to read an empty output buffer, or that data in the output buffer was lost.
3	DDE	The Device Dependent Error bit indicates that a device error occurred (such as a self-test error).
4	EXE	The Execution Error bit indicates that an error occurred when the device was executing a command or query.
5	CME	The Command Error bit indicates that a command syntax error occurred.
6	URQ	The User Request bit is not used by most instruments.
7	PON	The Power On bit indicates that the device is powered on.

For example, if you want to know when an execution error occurs, enable bit 4 of the ESER. Additionally, enable bit 5 of the SRER so that the error event of interest is reported by the ESB bit of the SBR.

Reading and Writing Register Information

This section describes the common GPIB commands used to read and write status and event register information.

Register Commands

Register	Operation	Command	Description
SESR	Read	`*ESR?`	Return a decimal value that corresponds to the weighted sum of all the bits set in the SESR register.
	Write	N/A	You cannot write to the SESR register.
ESER	Read	`*ESE?`	Return a decimal value that corresponds to the weighted sum of all the bits enabled by the `*ESE` command.
	Write	`*ESE`	Write a decimal value that corresponds to the weighted sum of all the bits you want to enable in the SESR register.
SBR	Read	`*STB?`	Return a decimal value that corresponds to the weighted sum of all the bits set in the SBR register. This command returns the same result as a serial poll except that the MSS bit is not cleared.
	Write	N/A	You cannot write to the SBR register.
SRER	Read	`*SRE?`	Return a decimal value that corresponds to the weighted sum of all the bits enabled by the `*SRE` command.
	Write	`*SRE`	Write a decimal value that corresponds to the weighted sum of all the bits you want to enable in the SBR register.

For example, to enable bit 4 of the SESR, you write the command `*ESE 16`. To enable bit 4 and bit 5 of the SESR, you write the command `*ESE 48`. To enable bit 5 of the SBR, you write the command `*SRE 32`.

To see how to use many of these commands in the context of an instrument control session, refer to “Execute Serial Poll” on page 5-29.

See Also

`visadev`

Related Examples

- “Get Started with GPIB Interface” on page 5-11
- “Write and Read GPIB Data” on page 4-10
- “Use Callbacks for VISA Communication” on page 5-24
- “Send Trigger Message” on page 5-27
- “Execute Serial Polls” on page 5-29

External Websites

- GPIB Instruments and MATLAB

Write and Read GPIB Data

In this section...
“Rules for Completing Write and Read Operations” on page 4-10
“Writing and Reading Text Data” on page 4-10
“Writing and Reading Binary Data” on page 4-12
“Parse Input String Data” on page 4-14

Rules for Completing Write and Read Operations

Completing Write Operations

A write operation using `write`, `writeline`, or `writebinblock` completes when one of these conditions is satisfied:

- The specified data is written.
- The time specified by the `Timeout` property passes.

An instrument determines if a write operation is complete based on the `Terminator` and `EOIMode` property values. The default value of `Terminator` is the line feed character. Refer to the documentation for your instrument to determine the terminator required by your instrument.

If `EOIMode` is `on`, then the End Or Identify (EOI) line is asserted when the last byte is written to the instrument. The last byte can be part of a binary data stream or a text data stream. The last byte written is the `Terminator` value and the EOI line is asserted when the instrument receives this byte.

Completing Read Operations

A read operation with `read`, `readline`, or `readbinblock` completes when one of these conditions is satisfied:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The specified number of values is read.

Writing and Reading Text Data

This example illustrates how to communicate with a VISA-GPIB instrument by writing and reading text data.

The instrument is a Tektronix TDS 210 two-channel oscilloscope. Therefore, many of the commands in the example are specific to this instrument. A sine wave is input into channel 2 of the oscilloscope, and you want to measure the peak-to-peak voltage of the input signal.

You can use these functions and properties when reading and writing text.

Function	Purpose
`writeline`	Write text to an instrument.

Function	Purpose
`readline`	Read data from an instrument and format as text.
NumBytesAvailable	Number of bytes available to read.
EOIMode	Whether EOI (end or identify) line is asserted.
Terminator	Character used to terminate commands sent to the instrument.

1 **Create a VISA-GPIB object** — Create the VISA-GPIB object `g` associated with a National Instruments GPIB controller with board index 0 and an instrument with primary address 1.

```
g = visadev("GPIB0::1::0::INSTR");
```

2 **Write and read data** — Write the `*IDN?` command to the instrument using `writeline`, and then read back the result of the command using `readline`.

```
writeline(g,"*IDN?")
g.NumBytesAvailable
```

```
ans =

     56
```

```
idn = readline(g)
```

```
idn = 

     "TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04"
```

You need to determine the measurement source. Possible measurement sources include channel 1 and channel 2 of the oscilloscope.

```
writeline(g,"MEASUREMENT:IMMED:SOURCE?")
source = readline(g)
```

```
source = 

     "CH1"
```

The scope is configured to return a measurement from channel 1. Because the input signal is connected to channel 2, you must configure the instrument to return a measurement from this channel.

```
writeline(g,"MEASUREMENT:IMMED:SOURCE CH2")
writeline(g,"MEASUREMENT:IMMED:SOURCE?")
source = readline(g)
```

```
source = 

     "CH2"
```

You can now configure the scope to return the peak-to-peak voltage, and then request the value of this measurement.

```
writeline(g,"MEASUREMENT:MEAS1:TYPE PK2PK")
writeline(g,"MEASUREMENT:MEAS1:VALUE?")
```

Read back the result using the `readline` function.

```
ptop = readline(g)
```

```
ptop = 

    "2.0199999809E0"
```

3 **Disconnect and clean up** — Use `clear` to disconnect the instrument from the VISA-GPIB object `g` and to clear it from the MATLAB workspace when you are done working with it.

```
clear g
```

ASCII Write Properties

By default, the End or Identify (EOI) line is asserted when the last byte is written to the instrument. This behavior is controlled by the `EOIMode` property. When `EOIMode` is set to `on`, the EOI line is asserted when the last byte is written to the instrument. When `EOIMode` is set to `off`, the EOI line is not asserted when the last byte is written to the instrument.

Writing and Reading Binary Data

This example illustrates how you can download the TDS 210 oscilloscope screen display. The screen display data is saved to disk using the Windows bitmap format. This data provides a permanent record of your work, and is an easy way to document important signal and scope parameters.

You use these functions when reading and writing binary data.

Function	Purpose
`read`	Read binary data from an instrument.
`write`	Write binary data to an instrument.

Note When performing a read or write operation, think of the received data in terms of values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

1 **Create a VISA-GPIB object** — Create the VISA-GPIB object `g` associated with a National Instruments GPIB controller with board index 0 and an instrument with primary address 1.

```
g = visadev("GPIB0::1::0::INSTR");
```

2 **Configure timeout value** — Configure the timeout value to two minutes to account for slow data transfer.

```
g.Timeout = 120;
```

3 **Write and read data** — Configure the scope to transfer the screen display as a bitmap.

```
writeline(g,"HARDCOPY:PORT GPIB")
writeline(g,"HARDCOPY:FORMAT BMP")
writeline(g,"HARDCOPY START")
```

Transfer the data to the MATLAB workspace as unsigned 8-bit integers.

```
out = read(g,g.NumBytesAvailable,"uint8");
```

4 **Disconnect and clean up** — Use `clear` to disconnect the instrument from the VISA-GPIB object `g` and to clear it from the MATLAB workspace when you are done working with it.

```
clear g
```

Viewing Bitmap Data

Follow these steps to view the bitmap data.

1. Open a disk file.
2. Write the data to the disk file.
3. Close the disk file.
4. Read the data using the `imread` function.
5. Scale and display the data using the `imagesc` function.

Use the MATLAB software file I/O functions `fopen`, `fwrite`, and `fclose`.

```
fid = fopen("test1.bmp","w");
fwrite(fid,out,"uint8");
fclose(fid)
a = imread("test1.bmp","bmp");
```

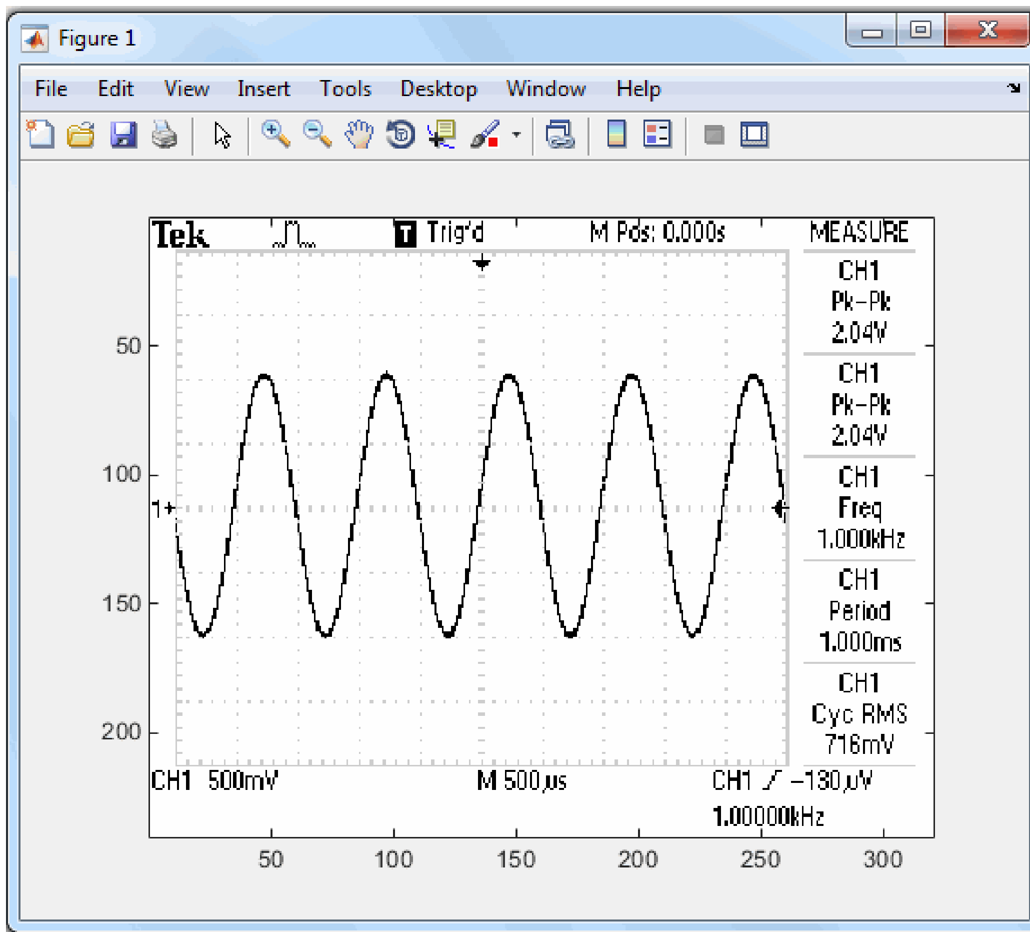
Display the image.

```
imagesc(a)
```

Use a gray colormap since the instrument generates only grayscale images.

```
c = colormap(gray);
colormap(flipud(c));
```

The resulting bitmap image is shown in the following figure.



Parse Input String Data

This example illustrates how to use the `split` function to parse data that you read from a Tektronix TDS 210 oscilloscope. The `split` function is particularly useful when you want to parse a string into one or more array elements, where each element is determined to be either a double or a character vector.

1 **Create a VISA-GPIB object** — Create the VISA-GPIB object `g` associated with a National Instruments GPIB controller with board index 0 and an instrument with primary address 1.

```
g = visadev("GPIB0::1::0::INSTR");
```

2 **Write and read data** — Return identification information to separate elements of a cell array using the `","` delimiters.

```
writeline(g,"*IDN?")
idn = readline(g);
idn = split(idn,",")
```

```
idn = 

  4×1 string array
```

```
    "TEKTRONIX"
    "TDS 210"
    "0"
    "CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04"
```

3 **Disconnect and clean up** — Use `clear` to disconnect the instrument from the VISA-GPIB object `g` and to clear it from the MATLAB workspace when you are done working with it.

```
clear g
```

See Also

`visadev` | `write` | `writeline` | `read` | `readline`

Related Examples

- "Use Callbacks for VISA Communication" on page 5-24

Transition Your Code to VISA-GPIB Interface

The `gpib` function, its object functions, and its properties will be removed. Use the VISA-GPIB interface with `visadev` instead.

gpib Interface	visadev Interface	Example
`instrhwinfo`	`visadevlist`	"Discover GPIB Instruments" on page 4-16
`visa`	`visadev`	"Connect to GPIB Instrument" on page 4-17
`fwrite` and `fread`	`write` and `read`	"Write and Read Binary or String Data" on page 4-17
`fprintf`	`writeline`	"Read Terminated String" on page 4-18
`fscanf`, `fgetl`, and `fgets`	`readline`	"Read Terminated String" on page 4-18
		"Read and Parse String Data" on page 4-18
`query`	`writeread`	"Write and Read Back Data" on page 4-19
`binblockwrite` and `binblockread`	`writebinblock` and `readbinblock`	"Write and Read Binblock Data" on page 4-19
`flushinput`, `flushoutput`, and `clrdevice`	`flush`	"Flush Data from Memory" on page 4-20
`Terminator`	`configureTerminator`	"Set Terminator" on page 4-20
`BytesAvailableFcnCount`, `BytesAvailableFcnMode`, `BytesAvailableFcn`, and `BytesAvailable`	`configureCallback`	"Set Up Callback Function" on page 4-20
`spoll`	`visastatus`	
`trigger`	`visatrigger`	
`gpib` Properties	`visadev` Properties on page 24-394	

Discover GPIB Instruments

This example shows how to discover GPIB instruments using the recommended functionality.

Functionality	Use This Instead
`instrhwinfo('gpib','ni')`	`list = visadevlist;` `list.ResourceName(list.Type=="gpib")`

For more information, see `visadevlist`.

Connect to GPIB Instrument

These examples show how to connect to a GPIB instrument and disconnect from it using the recommended functionality.

Functionality	Use This Instead
`g = gpib('ni',0,1)` `fopen(g)`	`g = visadev("GPIB0::1::0::INSTR");`

`fclose(g)`
`delete(g)`
`clear g`
	`clear g`

For more information, see `visadev`.

Write and Read Binary or String Data

These examples show how to perform a binary write and read, and how to write and read nonterminated string data, using the recommended functionality.

Functionality	Use This Instead

`% g is a gpibobject`
`fwrite(g,1:5)`
`data = fread(g,5)`
 	
`data =`
 	
`1`
`2`
`3`
`4`
`5`
	`% g is a visadev object`
	`write(g,1:5)`
	`data = read(g,5)`
	
	`data =`
	
	`1     2     3     4     5`

`% g is a gpib object`
`fwrite(g,'hello','char')`
`length = 5;`
`data = fread(g,length,'char')`
 	
`data =`
 	
`104`
`101`
`108`
`108`
`111`
 	
`data = char(data)'`
 	
`data =`
 	
`'hello'`
	`% g is a visadev object`
	`write(g,"hello","string")`
	`length = 5;`
	`data = read(g,length,"string")`
	
	`data =`
	
	`"hello"`

For more information, see `write` or `read`.

Read Terminated String

This example shows how to perform a terminated string write and read using the recommended functionality.

Functionality	Use This Instead

`% g is a gpib object`
`g.Terminator = 'CR/LF';`
`fprintf(g,'SOUR:%d:FREQ',ch)`
`data = fscanf(g,'%e')`
 	
`data =`
 	
`    11.9000`
	`% g is a visadev object`
	`configureTerminator(g,"CR/LF")`
	`str = sprintf("SOUR:%d:FREQ",ch)`
	`writeline(g,str)`
	`data = readline(g)`
	
	`data =`
	
	`    "11.9000"`
	
	`data = sscanf(data,'%e')`
	
	`data =`
	
	`    11.9000`

`% g is a gpib object`
`g.Terminator = 'CR/LF';`
`fprintf(g,'hello')`
`data = fgetl(g)`
 	
`data =`
 	
`    'hello'`
 	
`fgetl` reads until the specified terminator is reached and then discards the terminator.
	`% g is a visadev object`
	`configureTerminator(g,"CR/LF")`
	`writeline(g,"hello")`
	`data = readline(g)`
	
	`data =`
	
	`    "hello"`

`% g is a gpib object`
`g.Terminator = 'CR/LF';`
`fprintf(g,'hello')`
`data = fgets(g)`
 	
`data =`
 	
`    'hello`
`     '`
 	
`fgets` reads until the specified terminator is reached and then returns the terminator.	

For more information, see `writeline` or `readline`.

Read and Parse String Data

This example shows how to read and parse string data using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>% g is a gpib object
data = scanstr(g,';')

data =

  3×1 cell array

    {'a'}
    {'b'}
    {'c'}</pre></td><td><pre>% g is a visadev object
data = readline(g)

data =

    "a;b;c"

data = strsplit(data,";")

data =

  1×3 string array

    "a"   "b"   "c"</pre></td></tr>
</table>

For more information, see `readline`.

Write and Read Back Data

This example shows how to write ASCII terminated data and read ASCII terminated data back using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>% g is a gpib object
data = query(g,'ctrlcmd')

data =

    'success'</pre></td><td><pre>% g is a visadev object
data = writeread(g,"ctrlcmd")

data =

    "success"</pre></td></tr>
</table>

For more information, see `writeline` or `readline`.

Write and Read Binblock Data

This example shows how to write data with the IEEE standard binary block protocol using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>% g is a gpib object
binblockwrite(g,1:5);
data = binblockread(g)

data =

     1
     2
     3
     4
     5</pre></td><td><pre>% g is a visadev object
writebinblock(g,1:5)
data = readbinblock(g)

data =

     1   2   3   4   5</pre></td></tr>
</table>

For more information, see `writebinblock` or `readbinblock`.

Flush Data from Memory

This example shows how to flush data from the buffer using the recommended functionality.

Functionality	Use This Instead

`% g is a gpib object`
`flushinput(g)`
`clrdevice(g)`
	`% g is a visadev object`
	`flush(g,"input")`

`% g is a gpib object`
`flushoutput(g)`
`clrdevice(g)`
	`% g is a visadev object`
	`flush(g,"output")`

`% g is a gpib object`
`flushinput(g)`
`flushoutput(g)`
`clrdevice(g)`
	`% g is a visadev object`
	`flush(g)`

For more information, see `flush`.

Set Terminator

These examples show how to set the terminator using the recommended functionality.

Functionality	Use This Instead
`% g is a gpib object` `g.Terminator = "CR/LF";`	`% g is a visadev object` `configureTerminator(g,"CR/LF")`
`% g is a gpib object` `g.Terminator = {"CR/LF" [10]};`	`% g is a visadev object` `configureTerminator(g,"CR/LF",10)`

For more information, see `configureTerminator`.

Set Up Callback Function

These examples show how to set up a callback function using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>% g is a gpib object
g.BytesAvailableFcnCount = 5
g.BytesAvailableFcnMode = "byte"
g.BytesAvailableFcn = @mycallback

function mycallback(src,evt)
   data = fread(src,src.BytesAvailableFcnCount);
   disp(evt)
   disp(evt.Data)
end

    Type: 'BytesAvailable'
    Data: [1×1 struct]

    AbsTime: [2019 12 21 16 35 9.7032]</pre></td><td><pre>% g is a visadev object
configureCallback(g,"byte",5,@mycallback);
function mycallback(src,evt)
   data = read(src,src.BytesAvailableFcnCount);
   disp(evt)
end

  ByteAvailableInfo with properties:

    BytesAvailableFcnCount: 5
                   AbsTime: 21-Dec-2019 12:23:01</pre></td></tr>
<tr><td><pre>% g is a gpib object
g.Terminator = "CR"
g.BytesAvailableFcnMode = "terminator"
g.BytesAvailableFcn = @mycallback

function mycallback(src,evt)
   data = fscanf(src);
   disp(evt)
   disp(evt.Data)
end

    Type: 'BytesAvailable'
    Data: [1×1 struct]

    AbsTime: [2019 12 21 16 35 9.7032]</pre></td><td><pre>% g is a visadev object
configureTerminator(g,"CR")
configureCallback(g,"terminator",@mycallback)

function mycallback(src,evt)
   data = readline(src);
   disp(evt)
end

  TerminatorAvailableInfo with properties:

                   AbsTime: 21-Dec-2019 12:23:01</pre></td></tr>
</table>

For more information, see `configureCallback`.

See Also

`visadev`

More About

- R2021a GPIB Interface Topics

Controlling Instruments Using VISA

This chapter describes specific issues related to controlling instruments that use the VISA standard.

- “Get Started with VISA” on page 5-2
- “Get Started with TCP/IP Interface for VXI-11 and HiSLIP” on page 5-5
- “Get Started with TCP/IP Socket Interface” on page 5-7
- “Get Started with USB Interface” on page 5-9
- “Get Started with GPIB Interface” on page 5-11
- “Get Started with Serial Port Interface” on page 5-13
- “Get Started with VXI and PXI Interfaces” on page 5-15
- “Write and Read ASCII Data Using VISA” on page 5-19
- “Write and Read Binary Data Using VISA” on page 5-22
- “Use Callbacks for VISA Communication” on page 5-24
- “Send Trigger Message” on page 5-27
- “Execute Serial Polls” on page 5-29
- “Transition Your Code to visadev Interface” on page 5-32

Get Started with VISA

In this section...
“What Is VISA?” on page 5-2
“Supported Platforms and Minimum Driver Requirements” on page 5-2
“Interfaces Used with VISA” on page 5-2
“Connect to and Configure VISA Resource” on page 5-3

What Is VISA?

Virtual Instrument Standard Architecture (VISA) is a standard defined by Keysight (formerly Agilent Technologies®) and National Instruments for communicating with instruments regardless of the interface. The VISA standard was formerly maintained by the VXIplug&play Systems Alliance and is now maintained by the IVI Foundation.

Instrument Control Toolbox supports the TCP/IP (using VXI11 and HiSLIP), TCP/IP Socket, USB, GPIB, Serial, VXI, and PXI interfaces using the VISA standard. Communication is established through a VISA instrument object, which you create in the MATLAB workspace. For example, a VISA-GPIB object allows you to use the VISA standard to communicate with an instrument that possesses a GPIB interface.

For the full VISA specifications maintained by the IVI Foundation, see IVI Specifications.

Supported Platforms and Minimum Driver Requirements

VISA is supported on these platforms:

- macOS (NI-VISA and R&S VISA only)
- Windows 10

These are the minimum VISA driver versions you must have:

- Keysight IO Libraries version 18.1.24715.0 (Keysight Connection Expert 2019)
- National Instruments NI-VISA version 19.5
- Rohde & Schwarz R&S VISA version 5.12

Tektronix TekVISA is not supported for the `visadev` interface.

Interfaces Used with VISA

For many VISA applications, you can communicate with your instrument without detailed knowledge of how the interface works.

- “Get Started with TCP/IP Interface for VXI-11 and HiSLIP” on page 5-5
- “Get Started with TCP/IP Socket Interface” on page 5-7
- “Get Started with USB Interface” on page 5-9
- “Get Started with GPIB Interface” on page 5-11

- “Get Started with Serial Port Interface” on page 5-13
- “Get Started with VXI and PXI Interfaces” on page 5-15

Connect to and Configure VISA Resource

See a list of VISA resources available to connect to using `visadevlist`. This function provides a list of resource names and aliases. You can also find a device's resource name or alias from your VISA vendor's control software. For more information about the VISA resource name, see ResourceName.

To connect to a VISA resource, specify its resource name or alias using `visadev`.

After you connect to your instrument or device, you can configure its properties. For a full list of `visadev` properties and information about how to configure them, see visadev Properties.

Other Functionality

Use the following functions to communicate with the `visadev` object.

read	Read data from VISA resource
readline	Read line of ASCII string data from VISA resource
readbinblock	Read one binblock of data from VISA resource
write	Write data to VISA resource
writeline	Write line of ASCII data to VISA resource
writebinblock	Write one binblock of data to VISA resource
writeread	Write command to VISA resource and read response
configureTerminator	Set terminator for ASCII string communication with VISA resource
configureCallback	Set callback function and trigger condition for communication with VISA resource
flush	Clear buffers for communication with VISA resource
visastatus	Check status of VISA resource
visatrigger	Send trigger message to GPIB or VXI instruments
setDTR	Set serial DTR pin
setRTS	Set serial RTS pin
getpinstatus	Get serial pin status

See Also

`visadevlist` | `visadev`

Related Examples

- “Get Started with TCP/IP Interface for VXI-11 and HiSLIP” on page 5-5
- “Get Started with TCP/IP Socket Interface” on page 5-7
- “Get Started with USB Interface” on page 5-9
- “Get Started with GPIB Interface” on page 5-11
- “Get Started with Serial Port Interface” on page 5-13
- “Get Started with VXI and PXI Interfaces” on page 5-15
- “Write and Read ASCII Data Using VISA” on page 5-19
- “Write and Read Binary Data Using VISA” on page 5-22

External Websites

- IVI Specifications

Get Started with TCP/IP Interface for VXI-11 and HiSLIP

The TCP/IP interface is supported through a VISA-TCP/IP object. The features associated with a VISA-TCP/IP object are similar to the features associated with a `tcpclient` object. Therefore, only functions and properties that are unique to VISA's TCP/IP interface are discussed in this section. Both VXI-11 and HiSLIP protocols are supported.

Refer to "TCP/IP Communication Overview" on page 7-2 for more information about TCP/IP communication.

Create VISA-TCP/IP Object

Create a VISA-TCP/IP object with the `visadev` function. Each object is associated with an instrument connected to your computer.

`visadev` requires the resource name or alias as an input. The resource name consists of the name of the TCP/IP board index, IP address or host name, and LAN device name of your instrument. You can find the VISA-TCP/IP resource name or alias for a given instrument with the configuration tool provided by your vendor or with the `visadevlist` function. Define the alias using your VISA vendor configuration tool.

The VISA-TCP/IP resource name has the format
`TCPIP[`*`board`*`]::`*`remote_host`*`[::`*`lan_device_name`*`]::INSTR`.

For example, use the VISA-TCP/IP interface to connect to an instrument at IP address 169.254.2.20 using the VXI-11 protocol.

```
visatcpip = visadev("TCPIP0::169.254.2.20::inst0::INSTR")
```

```
visatcpip = 

  TCPIP with properties:

         ResourceName: "TCPIP0::169.254.2.20::inst0::INSTR"
                Alias: "Keysight_33210A"
               Vendor: "Agilent Technologies"
                Model: "33210A"
              LANName: "inst0"
    InstrumentAddress: "169.254.2.20"
    NumBytesAvailable: 0

  Show all properties, functions
```

The VISA-TCP/IP object `visatcpip` represents a connection to your instrument. Click `properties` in the object display to see a full list of VISA-TCP/IP properties.

```
         ResourceName: "TCPIP0::169.254.2.20::inst0::INSTR"
                Alias: "Keysight_33210A"
               Vendor: "Agilent Technologies"
                Model: "33210A"
              LANName: "inst0"
    InstrumentAddress: "169.254.2.20"
    NumBytesAvailable: 0
```

```
             SerialNumber: "MY57003523"
                     Type: tcpip
            PreferredVisa: "National Instruments VISA"

               BoardIndex: 0

                ByteOrder: "little-endian"
                  Timeout: 10
               Terminator: "LF"

    BytesAvailableFcnMode: "off"
   BytesAvailableFcnCount: 64
        BytesAvailableFcn: []
          NumBytesWritten: 0

         ErrorOccurredFcn: []
                 UserData: []
```

You can use dot notation to configure and display property values. For more information about configuring these properties, see visadev Properties.

You can communicate with your instrument using the `visadev` “Object Functions” on page 24-394.

See Also

`visadevlist` | `visadev`

Related Examples

- visadev Properties

Get Started with TCP/IP Socket Interface

Create VISA-Socket Object

Create a VISA-Socket object with the `visadev` function. Each object is associated with an instrument connected to your computer.

`visadev` requires the resource name or alias as an input. The resource name consists of the name of the TCP/IP board index, IP address or host name, and port of your instrument. You can find the VISA-Socket resource name or alias for a given instrument with the configuration tool provided by your vendor or with the `visadevlist` function. Define the alias using your VISA vendor configuration tool.

The VISA-Socket resource name has the format `TCPIP[`*`board`*`]::`*`remote_host`*`::`*`port`*`::SOCKET`.

For example, use the VISA-Socket interface to connect to a Keysight N9010B EXA Signal Analyzer.

```
visasocket = visadev("TCPIP0::A-N9010B-21026.local::5005::SOCKET")
```

```
visasocket = 

  Socket with properties:

        ResourceName: "TCPIP0::A-N9010B-21026.local::5005::SOCKET"
               Alias: "SIGNAL_ANALYZER"
              Vendor: "Keysight"
               Model: "N9010B"
           IPAddress: "A-N9010B-21026.local"
                Port: 5005
    NumBytesAvailable: 0

  Show all properties, functions
```

The VISA-Socket object `visasocket` represents a connection to your instrument. Click `properties` in the object display to see a full list of VISA-Socket properties.

```
             ResourceName: "TCPIP0::A-N9010B-21026.local::5005::SOCKET"
                    Alias: "SIGNAL_ANALYZER"
                   Vendor: "Keysight"
                    Model: "N9010B"
                IPAddress: "A-N9010B-21026.local"
                     Port: 5005
        NumBytesAvailable: 0

             SerialNumber: "314159265"
                     Type: socket
            PreferredVisa: "National Instruments VISA"

                ByteOrder: "little-endian"
                  Timeout: 10
               Terminator: "LF"

    BytesAvailableFcnMode: "off"
```

```
    BytesAvailableFcnCount: 64
         BytesAvailableFcn: []
           NumBytesWritten: 0

          ErrorOccurredFcn: []
                  UserData: []
```

You can use dot notation to configure and display property values. For more information about configuring these properties, see visadev Properties.

You can communicate with your instrument using the `visadev` “Object Functions” on page 24-394.

See Also

`visadevlist` | `visadev`

Related Examples

- visadev Properties

Get Started with USB Interface

Create VISA-USB Object

Create a VISA-USB object with the `visadev` function. Each object is associated with an instrument connected to a USB port on your computer.

`visadev` requires the resource name or alias as an input. The resource name consists of the name of the USB board index, vendor ID, product ID, serial number, and interface number of the connected instrument. You can find the VISA-USB resource name or alias for a given instrument with the configuration tool provided by your vendor or with the `visadevlist` function. Define the alias using your VISA vendor configuration tool.

The VISA-USB resource name has the format `USB[`*`board`*`]::`*`vendor_ID`*`::`*`product_ID`*`::`*`serial_number`*`[::`*`interface_number`*`]::INSTR`.

For example, use the VISA-USB interface to connect to a Tektronix TDS2024B digital oscilloscope.

```
visausb = visadev("USB0::0x0699::0x036A::CU010105::0::INSTR")
```

```
visausb = 

  USB with properties:

         ResourceName: "USB0::0x0699::0x036A::CU010105::0::INSTR"
                Alias: "NI_SCOPE_4CH"
               Vendor: "TEKTRONIX"
                Model: "TDS 2024B"
    NumBytesAvailable: 0

  Show all properties, functions
```

The VISA-USB object `visausb` represents a connection to your instrument. Click `properties` in the object display to see a full list of VISA-USB properties.

```
         ResourceName: "USB0::0x0699::0x036A::CU010105::0::INSTR"
                Alias: "NI_SCOPE_4CH"
               Vendor: "TEKTRONIX"
                Model: "TDS 2024B"
    NumBytesAvailable: 0

                 Type: usb
        PreferredVisa: "National Instruments VISA"

             VendorID: "0x0699"
            ProductID: "0x036A"
         SerialNumber: "CU010105"
           BoardIndex: 0
       InterfaceIndex: 0

            ByteOrder: "little-endian"
              Timeout: 10
           Terminator: "LF"
```

```
     BytesAvailableFcnMode: "off"
    BytesAvailableFcnCount: 64
         BytesAvailableFcn: []
           NumBytesWritten: 0

          ErrorOccurredFcn: []
                  UserData: []
```

You can use dot notation to configure and display property values. For more information about configuring these properties, see visadev Properties.

You can communicate with your instrument using the `visadev` “Object Functions” on page 24-394.

See Also

`visadevlist` | `visadev`

Related Examples

- visadev Properties

Get Started with GPIB Interface

You can communicate with GPIB instruments in MATLAB by using the VISA-GPIB interface.

Install Required Drivers

In order to use the VISA-GPIB interface, you must have the required drivers installed for both GPIB and VISA. The VISA-GPIB interface is supported on Windows 10 and not available for macOS or Linux.

The following table shows the minimum GPIB and VISA driver versions you must have. You must have one of the following GPIB drivers and its corresponding VISA driver both installed.

<table>
<tr><th>Minimum GPIB driver</th><th>Minimum VISA driver</th></tr>
<tr><td>Keysight IO Libraries version 18.1.24715.0 (Keysight Connection Expert 2019)</td><td rowspan="3">Keysight IO Libraries version 18.1.24715.0 (Keysight Connection Expert 2019)</td></tr>
<tr><td>ICS 488.2v4 Adaptor version 4.0</td></tr>
<tr><td>ADLINK ADL-GPIB version 20.01.0</td></tr>
<tr><td>NI-488.2 Adaptor v2.8</td><td rowspan="2">National Instruments NI-VISA version 19.5</td></tr>
<tr><td>MCC GPIB 488.2 Library v2.3</td></tr>
</table>

Create VISA-GPIB Object

Create a VISA-GPIB object with the `visadev` function. Each VISA-GPIB object is associated with:

- A GPIB controller installed in your computer
- An instrument with a GPIB interface

`visadev` requires the resource name or alias as an input. The resource name consists of the GPIB board index, the instrument primary address, and the instrument secondary address. You can find the VISA-GPIB resource name or alias for a given instrument with the configuration tool provided by your vendor or with the `visadevlist` function. Define the alias using your VISA vendor configuration tool.

The VISA-GPIB resource name has the format
`GPIB[`*`board`*`]::`*`primary_address`*`[::`*`secondary_address`*`]::INSTR`.

For example, use the VISA-GPIB interface to connect to a National Instruments controller with board index 0 and a Tektronix TDS1002 digital oscilloscope with primary address 1 and secondary address 0.

```
visagpib = visadev("GPIB0::1::0::INSTR")
```

```
visagpib = 

  GPIB with properties:

         ResourceName: "GPIB0::1::0::INSTR"
                Alias: "OSCOPE_2CH"
               Vendor: "TEKTRONIX"
```

```
                Model: "TDS 1002"
           BoardIndex: 0
       PrimaryAddress: 1
     SecondaryAddress: 0
    NumBytesAvailable: 0

  Show all properties, functions
```

The VISA-GPIB object `visagpib` represents a connection to your instrument. Click `properties` in the object display to see a full list of VISA-GPIB properties.

```
             ResourceName: "GPIB0::1::0::INSTR"
                    Alias: "OSCOPE_2CH"
                   Vendor: "TEKTRONIX"
                    Model: "TDS 1002"
               BoardIndex: 0
           PrimaryAddress: 1
         SecondaryAddress: 0
        NumBytesAvailable: 0

             SerialNumber: "0"
                     Type: gpib
            PreferredVisa: "National Instruments VISA"

                ByteOrder: "little-endian"
                  Timeout: 10
               Terminator: "LF"
                  EOIMode: on

     BytesAvailableFcnMode: "off"
    BytesAvailableFcnCount: 64
         BytesAvailableFcn: []
           NumBytesWritten: 0

          ErrorOccurredFcn: []
                  UserData: []
```

You can use dot notation to configure and display property values. For more information about configuring these properties, see visadev Properties.

You can communicate with your instrument using the `visadev` “Object Functions” on page 24-394.

See Also

`visadevlist` | `visadev`

Related Examples

- visadev Properties
- “Write and Read GPIB Data” on page 4-10

Get Started with Serial Port Interface

The serial port interface is supported through a VISA-Serial object. The features associated with a VISA-Serial object are similar to the features associated with a `serialport` object. Therefore, only functions and properties that are unique to VISA's serial port interface are discussed in this section.

Refer to "Serial Port Overview" on page 6-2 for more information about serial port communication.

Create VISA-Serial Object

Create a VISA-Serial object with the `visadev` function. Each object is associated with an instrument connected to a serial port on your computer.

`visadev` requires the resource name or alias as an input. The resource name consists of the name of the serial port connected to your instrument. You can find the VISA-Serial resource name or alias for a given instrument with the configuration tool provided by your vendor or with the `visadevlist` function. Define the alias using your VISA vendor configuration tool.

The VISA-Serial resource name has the format `ASRL[`*`port_number`*`]::INSTR`.

For example,use the VISA-Serial interface to connect to the COM1 port, use the following command.

```
visaserial = visadev("ASRL1::INSTR")
```

```
visaserial = 

  Serial with properties:

         ResourceName: "ASRL1::INSTR"
                Alias: "COM1"
                 Port: "ASRL1"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

The VISA-Serial object `visaserial` represents a connection to your instrument. Click `properties` in the object display to see a full list of VISA-Serial properties.

```
         ResourceName: "ASRL1::INSTR"
                Alias: "COM1"
                 Port: "ASRL1"
             BaudRate: 9600
    NumBytesAvailable: 0

                 Type: serial
        PreferredVisa: "National Instruments VISA"

            ByteOrder: "little-endian"
             DataBits: 8
             StopBits: 1
               Parity: none
          FlowControl: none
              Timeout: 10
```

```
              Terminator: "LF"

     BytesAvailableFcnMode: "off"
    BytesAvailableFcnCount: 64
         BytesAvailableFcn: []
           NumBytesWritten: 0

          ErrorOccurredFcn: []
                  UserData: []
```

You can use dot notation to configure and display property values. For more information about configuring these properties, see visadev Properties.

You can communicate with your instrument using the `visadev` “Object Functions” on page 24-394.

Configure Communication Settings

Before you can write or read data, both the VISA-Serial object and the instrument must have identical communication settings. Configuring serial port communications involves specifying values for properties that control the baud rate and the “Serial Data Format” on page 6-6. These properties are as follows.

VISA-Serial Communication Properties

Property Name	Description
BaudRate	Specify the rate at which bits are transmitted.
DataBits	Specify the number of data bits to transmit.
Parity	Specify the type of parity checking.
StopBits	Specify the number of bits used to indicate the end of a byte.
Terminator	Specify the character used to terminate commands written to the instrument.

Refer to your instrument documentation for an explanation of its supported communication settings.

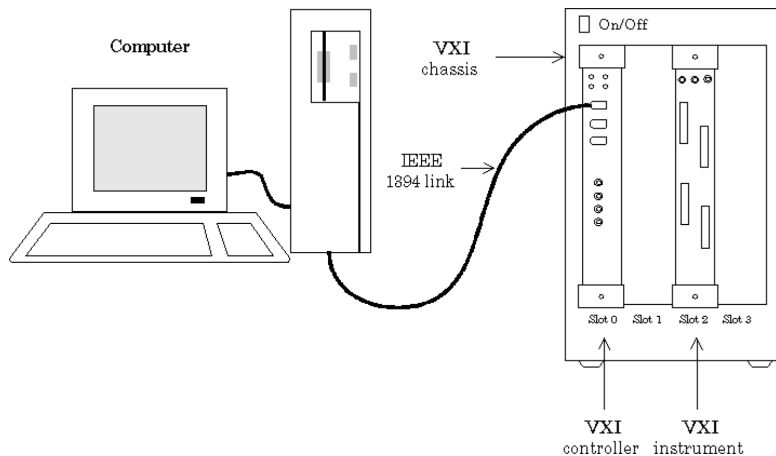
See Also
`visadevlist` | `visadev`

Related Examples

- visadev Properties

Get Started with VXI and PXI Interfaces

The VXI interface is associated with a VXI controller that you install in slot 0 of a VXI chassis. This interface, along with the other relevant hardware, is shown below.



The VXI interface is supported through a VISA-VXI object. Many of the features associated with a VISA-VXI object are similar to the features associated with other instrument objects. Therefore, only functions and properties that are unique to VISA's VXI interface are discussed in this section.

A PXI interface is supported through a VISA-PXI object. Features associated with a VISA-PXI object are identical to the features associated with a VISA-VXI object.

PXI devices may be supported by other toolboxes or come with higher level drivers that are easier to interact with than the raw PXI interface.

Create VISA-VXI Object

Create a VISA-VXI object with the `visadev` function. Each object is associated with:

- A VXI chassis
- A VXI controller in slot 0 of the VXI chassis
- An instrument installed in the VXI chassis

`visadev` requires the resource name or alias as an input. The resource name consists of the VXI chassis index and the instrument logical address. You can find the VISA-VXI resource name or alias for a given instrument with the configuration tool provided by your vendor or with the `visadevlist` function. Define the alias using your VISA vendor configuration tool.

The VISA-VXI resource name has the format `VXI[`*`chassis`*`]::`*`VXI_logical_address`*`::INSTR`.

For example, use the VISA-VXI interface to connect to a VXI chassis with index 0 and a Keysight E1432A 16-channel digitizer with logical address 32.

```
visavxi = visadev("VXI0::32::INSTR")
```

```
visavxi =
```

```
  VXI with properties:

         ResourceName: "VXI0::32::INSTR"
                Alias: "DIGITIZER_16CH"
               Vendor: "Keysight"
                Model: "E1432A"
    NumBytesAvailable: 0

  Show all properties, functions
```

The VISA-VXI object `visavxi` represents a connection to your instrument. Click `properties` in the object display to see a full list of VISA-VXI properties.

```
              ResourceName: "VXI0::32::INSTR"
                     Alias: "DIGITIZER_16CH"
                    Vendor: "Keysight"
                     Model: "E1432A"
         NumBytesAvailable: 0

              SerialNumber: "P12345"
                      Type: vxi
             PreferredVisa: "National Instruments VISA"

              ChassisIndex: 0
            LogicalAddress: 32
                      Slot: 0

                 ByteOrder: "little-endian"
                   Timeout: 10
                Terminator: "LF"
                   EOIMode: on

      BytesAvailableFcnMode: "off"
     BytesAvailableFcnCount: 64
          BytesAvailableFcn: []
            NumBytesWritten: 0

           ErrorOccurredFcn: []
                   UserData: []
```

You can use dot notation to configure and display property values. For more information about configuring these properties, see visadev Properties.

You can communicate with your instrument using the `visadev` “Object Functions” on page 24-394.

Create VISA-PXI Object

Create a VISA-PXI object with the `visadev` function. Each object is associated with:

- A PXI chassis
- A PXI controller in slot 0 of the PXI chassis
- An instrument installed in the PXI chassis

`visadev` requires the resource name or alias as an input. The resource name consists of the PCI bus number, PCI device number, PCI function number, PXI chassis index, and slot number. You can find

the VISA-PXI resource name or alias for a given instrument with the configuration tool provided by your vendor or with the `visadevlist` function. Define the alias using your VISA vendor configuration tool.

The VISA-PXI resource name has the format `PXI[`*bus*`]::`*device*`[::`*function*`][::INSTR]` or `PXI[`*bus*`]::CHASSIS`*chassis*`::SLOT`*slot*`[::FUNC`*function*`][::INSTR]`.

For example, use the VISA-PXI interface to connect to a Keysight E1432A 16-channel digitizer with logical address 32.

```
visapxi = visadev("PXI0::1::2::INSTR")
```

```
visapxi = 

  PXI with properties:

         ResourceName: "PXI0::1::2::INSTR"
                Alias: "DIGITIZER_16CH"
               Vendor: "Keysight"
                Model: "E1432A"
    NumBytesAvailable: 0

  Show all properties, functions
```

The VISA-PXI object `visapxi` represents a connection to your instrument. Click `properties` in the object display to see a full list of VISA-PXI properties.

```
              SerialNumber: "P67890"
                      Type: pxi
             PreferredVisa: "National Instruments VISA"

                       Bus: 0
               DeviceIndex: 1
             FunctionIndex: 2
                      Slot: 0

                 ByteOrder: "little-endian"
                   Timeout: 10
                Terminator: "LF"
                   EOIMode: on

     BytesAvailableFcnMode: "off"
    BytesAvailableFcnCount: 64
         BytesAvailableFcn: []
           NumBytesWritten: 0

          ErrorOccurredFcn: []
                  UserData: []
```

You can use dot notation to configure and display property values. For more information about configuring these properties, see visadev Properties.

You can communicate with your instrument using the `visadev` "Object Functions" on page 24-394.

See Also

`visadevlist` | `visadev`

Related Examples

- visadev Properties

Write and Read ASCII Data Using VISA

This example explores ASCII read and write operations with a VISA object using a Tektronix TDS210 oscilloscope.

The VISA object supports seven interfaces: serial, GPIB, VXI, PXI, USB, Serial, TCP/IP, and Socket. This example explores ASCII read and write operations using a VISA-GPIB object. However, ASCII read and write operations for all interfaces are identical to each other. Therefore, you can use the same commands. The only difference is the resource name specified in the VISA constructor `visadev`.

ASCII read and write operations for the VISA-Serial object are identical to ASCII read and write operations for the serial port object. Therefore, to learn how to perform ASCII read and write operations for the VISA-Serial object, refer to “Write and Read Serial Port Data” on page 6-17.

Connect to Instrument

Create a VISA-GPIB object using the VISA resource string shown below.

```
v = visadev("GPIB0::2::INSTR")
```

```
v = 

  GPIB with properties:

         ResourceName: "GPIB0::2::INSTR"
                Alias: "OSCOPE"
               Vendor: "TEKTRONIX"
                Model: "TDS 210"
           BoardIndex: 0
       PrimaryAddress: 1
     SecondaryAddress: 0
    NumBytesAvailable: 0

  Show all properties, functions
```

Write ASCII Data

Use the `writeline` function to write ASCII data to the instrument. For example, the `"Display:Contrast"` command changes the display contrast of the oscilloscope.

```
writeline(v,"Display:Contrast 45")
```

The function suspends MATLAB execution until all the data is written or a timeout occurs as specified by the `Timeout` property of the `visadev` object.

Check the default ASCII terminator.

```
v.Terminator
```

```
ans = 

    "LF"
```

The `writeline` function automatically appends the linefeed (LF) terminator to `"Display:Contrast 45"` before it is written to the server, indicating the end of the command.

Check the value of the `EOIMode` property. This property is only available for VISA-GPIB, VISA-VXI, and VISA-PXI interfaces.

```
v.EOIMode
```

```
ans = 

  OnOffSwitchState enumeration

    on
```

By default, the End or Identify (EOI) line is asserted when the last byte is written to the instrument. This behavior is controlled by the `EOIMode` property. When `EOIMode` is set to `on`, the EOI line is asserted when the last byte is written to the instrument. When `EOIMode` is set to `off`, the EOI line is not asserted when the last byte is written to the instrument.

Confirm the success of the write operation by viewing the `NumBytesAvailable` property.

```
v.NumBytesWritten
```

```
ans = 

    20
```

Clear any data in the buffer before moving to the next step.

```
flush(v)
```

Read ASCII Data

Use the `readline` function to read ASCII data from the instrument. For example, the oscilloscope command `"Display:Contrast?"` returns the oscilloscope's display contrast.

```
writeline(v,"Display:Contrast?")
data = readline(v)
```

```
data = 

    45
```

The `readline` function reads data until it reaches a terminator, removes the terminator, and returns the data.

You can also use the `writeread` function to perform the same operation. Write an ASCII command to your instrument and read the response.

```
data = writeread(v,"Display:Contrast?")
```

```
data = 

    45
```

Clean Up

When you are finished with the VISA-GPIB object, clear it.

```
clear v
```

See Also

`visadev` | `readline` | `writeline` | `writeread`

Related Examples

- “Write and Read Binary Data Using VISA” on page 5-22

Write and Read Binary Data Using VISA

This example explores binary read and write operations with a VISA object using a Tektronix TDS210 oscilloscope.

The VISA object supports seven interfaces: serial, GPIB, VXI, PXI, USB, Serial, TCP/IP, and Socket. This example explores binary read and write operations using a VISA-GPIB object. However, binary read and write operations for all interfaces are identical to each other. Therefore, you can use the same commands. The only difference is the resource name specified in the VISA constructor `visadev`.

Binary read and write operations for the VISA-Serial object are identical to binary read and write operations for the serial port object. Therefore, to learn how to perform binary read and write operations for the VISA-Serial object, refer to “Write and Read Serial Port Data” on page 6-17.

Connect to Instrument

Create a VISA-GPIB object using the VISA resource string shown below.

```
v = visadev("GPIB0::2::INSTR")
```

```
v = 

  GPIB with properties:

         ResourceName: "GPIB0::2::INSTR"
                Alias: "OSCOPE"
               Vendor: "TEKTRONIX"
                Model: "TDS 210"
           BoardIndex: 0
       PrimaryAddress: 1
     SecondaryAddress: 0
    NumBytesAvailable: 0

  Show all properties, functions
```

Write Binary Data

Use the `write` function to write binary data to the instrument. The following commands configure and then send a sine wave to the instrument.

```
writeline(v,"Data:Destination RefB");
writeline(v,"Data:Encdg SRPbinary");
writeline(v,"Data:Width 2";
writeline(v,"Data:Start 1");

t = (0:499) .* 8 * pi / 500;
data = round(sin(t) * 90 + 127);
writeline(v,"CURVE #3500");

write(v,data,"int16")
```

The `write` function suspends MATLAB execution until all the data is written or a timeout occurs as specified by the `Timeout` property of the `visadev` object.

By default, the `write` function writes binary data as `uint8` data. For more information about specifying other data types, see `write`.

Note When performing a write operation, you should think of the transmitted data in terms of values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

Read Binary Data

Use the `read` function to read binary data from the instrument. Use the following commands to read the sine wave from the instrument.

```
writeline(v,"Data:Source CH1");
writeline(v,"Data:Encdg SRIbinary");
writeline(v,"Data:Width 2");
writeline(v,"Data:Start 1");
writeline(v,"Curve?")

data = read(v,1200,"int16");
```

The `read` function suspends MATLAB execution until one of the following occurs:

- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled
- The specified number of values is read
- The EOI line is asserted
- The terminator is received as specified by the `Terminator` property

By default, the `read` function reads binary data as `uint8` data. For more information about specifying other data types, see `read`.

Note When performing a read operation, you should think of the received data in terms of values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

Clean Up

When you are finished with the VISA-GPIB object, clear it.

```
clear v
```

See Also

`visadev` | `read` | `write`

Related Examples

- “Write and Read ASCII Data Using VISA” on page 5-19

Use Callbacks for VISA Communication

You can enhance the power and flexibility of your instrument control application by using events and callbacks. An event occurs after a condition is met and can result in one or more callbacks.

While MATLAB is connected to the instrument, you can use events to display a message, display data, analyze data, and so on. You can control callbacks through `configureCallback` and callback functions. Callback functions are MATLAB functions that you write to suit your specific application needs.

Use Events and Callbacks

This example uses the callback function `mycallback` to read from the instrument when a terminator is available to be read. The event is generated when the `Terminator` property value is read. Specify the event type and the callback function to be executed using the `configureCallback` function. Specify the callback function as a function handle.

```
function mycallback(src,evt)
   data = readline(src)
   disp(evt)
end

g = visadev("GPIB0::1::0::INSTR");
configureCallback(g,"terminator",@mycallback)
writeline(g,"*IDN?")
```

The resulting display from `mycallback` is shown below.

```
data = 

    "TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04"

  DataAvailableInfo with properties:

    BytesAvailableFcnCount: 1
                   AbsTime: 1-Apr-2021 14:54:16
```

End the VISA-GPIB session.

```
clear g
```

Event Types and Callback Properties

The following table lists the `visadev` properties and functions associated with callbacks.

Property or Function	Purpose
`configureCallback`	Set callback function and trigger condition for communication
`BytesAvailableFcn`	Callback function triggered by bytes available event
`BytesAvailableFcnCount`	Number of bytes of data to trigger callback

Property or Function	Purpose
BytesAvailableFcnMode	Bytes available callback trigger mode
ErrorOccurredFcn	Callback function triggered by error event
UserData	General purpose property for user data

For more information about configuring these properties and functions, see visadev Properties.

Bytes-Available Event Modes

A bytes-available event is generated immediately after a specified number of bytes are available in the input buffer or the terminator character specified is read, as determined by the `BytesAvailableFcnMode` property.

- If `BytesAvailableFcnMode` is `byte`, the bytes-available event executes the callback function specified for the `BytesAvailableFcn` property every time the number of bytes specified by `BytesAvailableFcnCount` is stored in the input buffer.
- If `BytesAvailableFcnMode` is `terminator`, the bytes-available event executes the callback function specified for the `BytesAvailableFcn` property every time the character specified by the `Terminator` property is read.

Error Event

An error event is generated immediately after an error occurs. An error event is generated when the connection to your VISA resource is interrupted or when an asynchronous read error occurs. An error event is not generated for configuration errors such as setting an invalid property value. This event executes the callback function specified for the `ErrorOccurredFcn` property.

Use Events and Callbacks to Display Event Information

This example extends “Writing and Reading Binary Data” on page 4-12 by using the custom callback function `mycallback` to display event-related information to the command line when a bytes-available event occurs during a binary read operation.

1. Create a callback function `mycallback` and save it as an .m file in the directory that you are working in.

   ```
   function mycallback(src,evt)
      disp(evt)
   end
   ```
2. Create the VISA-GPIB object `g` associated with a National Instruments GPIB controller with primary address 1 and secondary address 0.

   ```
   g = visadev("GPIB0::1::0::INSTR");
   ```
3. Configure the timeout value to two minutes to account for slow data transfer.

   ```
   g.Timeout = 120;
   ```

   Configure `g` to execute the callback function `mycallback` every time 5000 bytes is stored in the input buffer.

   ```
   configureCallback(g,"byte",5000,@mycallback)
   ```
4. Configure the scope to transfer the screen display as a bitmap.

```
writeline(g,"HARDCOPY:PORT GPIB")
writeline(g,"HARDCOPY:FORMAT BMP")
writeline(g,"HARDCOPY START")
```

`mycallback` is called every time 5000 bytes is stored in the input buffer. The resulting displays are as follows.

```
  DataAvailableInfo with properties:

    BytesAvailableFcnCount: 5000
                   AbsTime: 1-Apr-2021 15:06:11

  DataAvailableInfo with properties:

    BytesAvailableFcnCount: 5000
                   AbsTime: 1-Apr-2021 15:06:16

  DataAvailableInfo with properties:

    BytesAvailableFcnCount: 5000
                   AbsTime: 1-Apr-2021 15:06:21
```

5. After all the data is sent to the input buffer, transfer the data to the MATLAB workspace as unsigned 8-bit integers.

   ```
   out = read(g,g.NumBytesAvailable,"uint8");
   ```
6. Use `clear` to disconnect the instrument from the VISA-GPIB object `g` and to clear it from the MATLAB workspace when you are done working with it.

   ```
   clear g
   ```

See Also

`visadev`

Related Examples

- “Writing and Reading Binary Data” on page 4-12

Send Trigger Message

Use visatrigger Function

You can execute a trigger with the `visatrigger` function. This function is only for VISA-GPIB and VISA-VXI interfaces. It is equivalent to the `viAssertTrigger` operation, as described in the VISA Specifications found at IVI Specifications. For an instrument connected with the VISA-GPIB interface, this function sends the GPIB GET (Group Execute Trigger) command.

Refer to your instrument documentation to learn how to use its triggering capabilities.

Execute Trigger

This example illustrates VISA-GPIB triggering using a Keysight 33120A function generator. The output of the function generator is displayed with an oscilloscope so that you can observe the trigger.

1. **Create a VISA-GPIB object** — Create the VISA-GPIB object `g` associated with a National Instruments GPIB controller with board index 0 and an instrument with primary address 1.

   ```
   g = visadev("GPIB0::1::0::INSTR");
   ```
2. **Write and read data** — Configure the function generator to produce a 5000 Hz sine wave, with 6 volts peak-to-peak.

   ```
   writeline(g,"Func:Shape Sin")
   writeline(g,"Volt 3")
   writeline(g,"Freq 5000")
   ```

   Configure the burst of the trigger to display the sine wave for 5 seconds, configure the function generator to expect the trigger from the GPIB board, and enable the burst mode.

   ```
   writeline(g,"BM:NCycles 25000")
   writeline(g,"Trigger:Source Bus")
   writeline(g,"BM:State On")
   ```

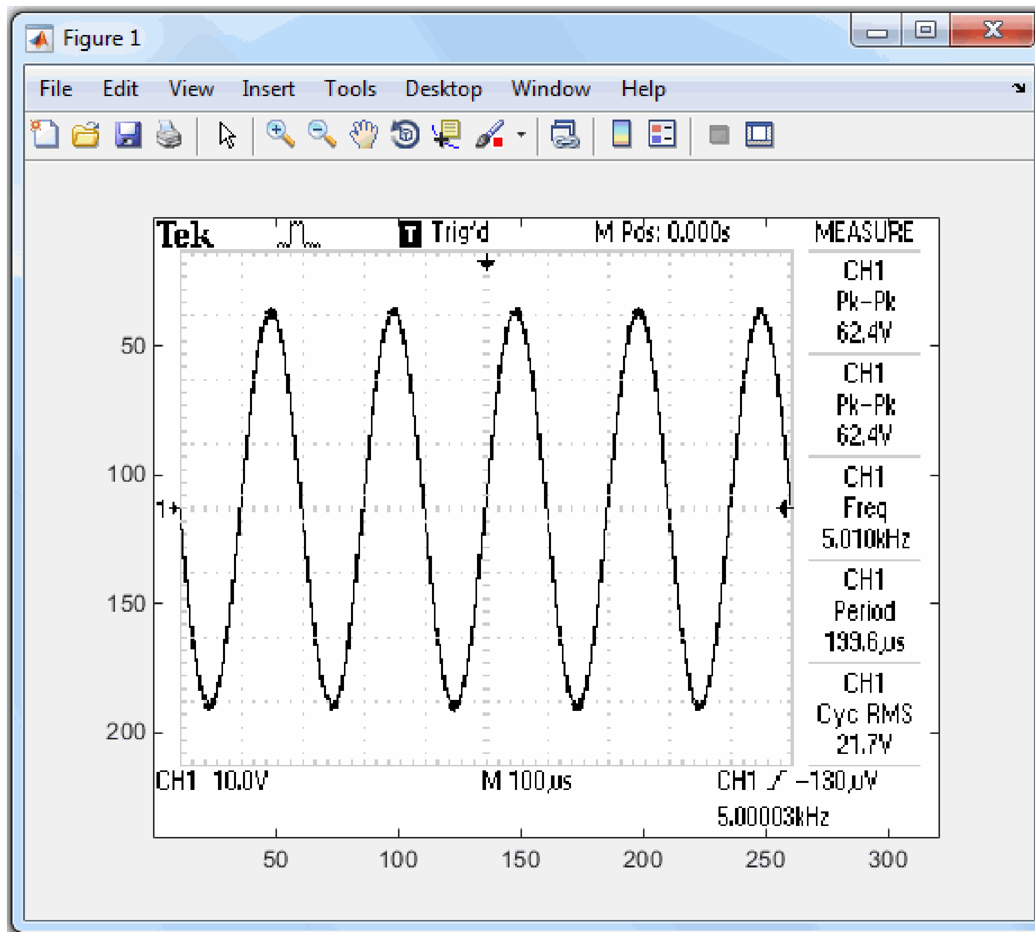
   Trigger the instrument.

   ```
   visatrigger(g)
   ```

   Disable the burst mode.

   ```
   writeline(g,"BM:State Off")
   ```

   While the function generator is triggered, the sine wave is saved to the Ref A memory location of the oscilloscope. The saved waveform is shown in the following figure.



3 **Disconnect and clean up** — Use `clear` to disconnect the instrument from the VISA-GPIB object `g` and to clear it from the MATLAB workspace when you are done working with it.

```
clear g
```

See Also

`visatrigger`

Related Examples

- "Write and Read ASCII Data Using VISA" on page 5-19
- "Write and Read GPIB Data" on page 4-10

Execute Serial Polls

Use visastatus Function

You can execute a serial poll with the `visastatus` function. This function is for all VISA interfaces.

Serial Poll for VISA-GPIB

In a serial poll for the VISA-GPIB interface, the Controller asks (polls) each addressed Listener to send back a status byte that indicates whether it has asserted the SRQ line and needs servicing. The seventh bit of this byte (the RQS bit) is set if the instrument is requesting service.

The Controller performs the following steps for every addressed Listener:

1. The Listener is addressed to talk and the Serial Poll Enable (SPE) command byte is sent.
2. The ATN line is set high and the Listener returns the status byte.
3. The ATN line is set low and the Serial Poll Disable (SPD) command byte is sent to end the poll sequence.

Refer to “Status and Event Reporting” on page 4-6 for more information on the GPIB bus lines and the RQS bit.

Execute Serial Poll

This example shows you how to execute a serial poll for a Keysight 33120A function generator and a Tektronix TDS 210 oscilloscope. In doing so, the example shows you how to configure many of the status bits described in “Standard Event Status Register” on page 4-8.

1. **Create VISA-GPIB objects** — Create the VISA-GPIB object `fgen` associated with a Keysight 33120A function generator at primary address 1.

   ```
   fgen = visadev("GPIB0::1::0::INSTR");
   ```

   Create the VISA-GPIB object `scope` associated with a Tektronix TDS 210 oscilloscope at primary address 2.

   ```
   scope = visadev("GPIB0::2::0::INSTR");
   ```
2. **Configure property values** — Configure both objects to time out after 1 second.

   ```
   fgen.Timeout = 1;
   scope.Timeout = 1;
   ```
3. **Write and read data** — Configure the function generator to request service when a command error occurs.

   ```
   writeline(fgen,"*CLS");
   writeline(fgen,"*ESE 32");
   writeline(fgen,"*SRE 32");
   ```

   Configure the oscilloscope to request service when a command error occurs.

   ```
   writeline(scope,"*CLS");
   writeline(scope,"*PSC 0");
   writeline(scope,"*ESE 32");
   ```

```
writeline(scope,"DESE 32");
writeline(scope,"*SRE 32");
```

Determine if any instrument needs servicing.

```
visastatus(fgen)


ans =

  logical

   0

visastatus(scope)


ans =

  logical

   0
```

Query the voltage value for each instrument.

```
writeline(fgen,"Volt?");
writeline(scope,"Volt?");
```

Determine if either instrument produced an error due to the preceding query.

```
visastatus(fgen)


ans =

  logical

   0

visastatus(scope)


ans =

  logical

   1
```

Since `Volt?` is a valid command for the function generator, the value is read back successfully.

```
volt1 = readline(fgen)


volt1 =

  +1.00000E-01
```

Since `Volt?` is an invalid command for the oscilloscope, it is requesting service. The oscilloscope read operation times out after 1 second.

```
volt2 = readline(scope)

Warning: The specified amount of data was not returned within the Timeout period for 'readlin
'visadev' unable to read any data. For more information on possible reasons, see visadev Read
Warnings.

ans =

     []
```

4 **Disconnect and clean up** — Use `clear` to disconnect the instruments from the VISA-GPIB objects `fgen` and `scope` and to clear it from the MATLAB workspace when you are done working with it.

```
clear fgen scope
```

See Also

`visastatus`

Related Examples

- "Write and Read GPIB Data" on page 4-10

Transition Your Code to visadev Interface

The `visa` function, its object functions, and its properties will be removed. Use `visadev` instead.

visa Interface	visadev Interface	Example
`instrhwinfo`	`visadevlist`	"Discover VISA Devices" on page 5-32
`visa`	`visadev`	"Connect to VISA Device" on page 5-33
`fwrite` and `fread`	`write` and `read`	"Write and Read Binary or String Data" on page 5-33
`fprintf`	`writeline`	"Read Terminated String" on page 5-34
`fscanf`, `fgetl`, and `fgets`	`readline`	"Read Terminated String" on page 5-34
		"Read and Parse String Data" on page 5-34
`query`	`writeread`	"Write and Read Back Data" on page 5-35
`binblockwrite` and `binblockread`	`writebinblock` and `readbinblock`	"Write and Read Binblock Data" on page 5-35
`flushinput`, `flushoutput`, and `clrdevice`	`flush`	"Flush Data from Memory" on page 5-36
`Terminator`	`configureTerminator`	"Set Terminator" on page 5-36
`BytesAvailableFcnCount`, `BytesAvailableFcnMode`, `BytesAvailableFcn`, and `BytesAvailable`	`configureCallback`	"Set Up Callback Function" on page 5-36
`spoll`	`visastatus`	
`trigger`	`visatrigger`	
`memmap`, `mempeek`, `mempoke`, `memread`, `memunmap`, `memwrite`	Not supported	
`visa` Properties	`visadev` Properties on page 24-394	

Discover VISA Devices

This example shows how to discover VISA devices using the recommended functionality.

Functionality	Use This Instead
`instrhwinfo('visa','ni')`	`visadevlist`

For more information, see `visadevlist`.

Connect to VISA Device

These examples show how to connect to a VISA device and disconnect from it using the recommended functionality.

Functionality	Use This Instead
v = visa('ni','GPIB::1::0::INSTR') fopen(v)	v = visadev("GPIB::1::0::INSTR");
fclose(v) delete(v) clear v	clear v

For more information, see `visadev`.

Write and Read Binary or String Data

These examples show how to perform a binary write and read, and how to write and read nonterminated string data, using the recommended functionality.

Functionality	Use This Instead

% v is a visa object
fwrite(v,1:5)
data = fread(v,5)
 	
data =
 	
     1
     2
     3
     4
     5
	% v is a visadev object
	write(v,1:5)
	data = read(v,5)
	
	data =
	
	     1 2 3 4 5

% v is a visa object
fwrite(v,"hello","char")
length = 5;
data = fread(v,length,"char")
 	
data =
 	
   104
   101
   108
   108
   111
 	
data = char(data)'
 	
data =
 	
    'hello'
	% v is a visadev object
	write(v,"hello","string")
	length = 5;
	data = read(v,length,"string")
	
	data =
	
	    "hello"

For more information, see `write` or `read`.

Read Terminated String

This example shows how to perform a terminated string write and read using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>% v is a visa object
v.Terminator = "CR";
fprintf(v,"hello")
data = fscanf(v)

data =

    'hello
     '</pre></td><td rowspan="3"><pre>% v is a visadev object
configureTerminator(v,"CR")
writeline(v,"hello")
data = readline(v)

a =

    "hello"</pre></td></tr>
<tr><td><pre>% v is a visa object
v.Terminator = "CR";
fprintf(v,"hello")
data = fgetl(v)

data =

    'hello'</pre>
<code>fgetl</code> reads until the specified terminator is reached and then discards the terminator.</td></tr>
<tr><td><pre>% v is a visa object
v.Terminator = "CR";
fprintf(v,"hello")
data = fgets(v)

data =

    'hello
     '</pre>
<code>fgets</code> reads until the specified terminator is reached and then returns the terminator.</td></tr>
</table>

For more information, see `writeline` or `readline`.

Read and Parse String Data

This example shows how to read and parse string data using the recommended functionality.

Functionality	Use This Instead

`% v is a visa object`
`data = scanstr(v,';')`
 	
`data =`
 	
`  3×1 cell array`
 	
`    {'a'}`
`    {'b'}`
`    {'c'}`
	`% v is a visadev object`
	`data = readline(v)`
	
	`data =`
	
	`    "a;b;c"`
	
	`data = strsplit(v,";")`
	
	`data =`
	
	`  1×3 string array`
	
	`    "a"    "b"    "c"`

For more information, see `readline`.

Write and Read Back Data

This example shows how to write ASCII terminated data and read ASCII terminated data back using the recommended functionality.

Functionality	Use This Instead

`% v is a visa object`
`data = query(v,'ctrlcmd')`
 	
`data =`
 	
`    'success'`
	`% v is a visadev object`
	`data = writeread(v,"ctrlcmd")`
	
	`data =`
	
	`    "success"`

For more information, see `writeline` or `readline`.

Write and Read Binblock Data

This example shows how to write data with the IEEE standard binary block protocol using the recommended functionality.

Functionality	Use This Instead

`% v is a visa object`
`binblockwrite(v,1:5);`
`data = binblockread(v)`
 	
`data =`
 	
`     1`
`     2`
`     3`
`     4`
`     5`
	`% v is a visadev object`
	`writebinblock(v,1:5)`
	`data = readbinblock(v)`
	
	`data =`
	
	`     1     2     3     4     5`

For more information, see `writebinblock` or `readbinblock`.

Flush Data from Memory

This example shows how to flush data from the buffer using the recommended functionality.

Functionality	Use This Instead

`% v is a visa object`
`flushinput(v)`
`clrdevice(v)`
	`% v is a visadev object`
	`flush(v,"input")`

`% v is a visa object`
`flushoutput(v)`
`clrdevice(v)`
	`% v is a visadev object`
	`flush(v,"output")`

`% v is a visa object`
`flushinput(v)`
`flushoutput(v)`
`clrdevice(v)`
	`% v is a visadev object`
	`flush(v)`

For more information, see `flush`.

Set Terminator

These examples show how to set the terminator using the recommended functionality.

Functionality	Use This Instead
`% v is a visa object` `v.Terminator = "CR/LF";`	`% v is a visadev object` `configureTerminator(v,"CR/LF")`
`% v is a visa object` `v.Terminator = {"CR/LF" [10]};`	`% v is a visadev object` `configureTerminator(v,"CR/LF",10)`

For more information, see `configureTerminator`.

Set Up Callback Function

These examples show how to set up a callback function using the recommended functionality.

Functionality	Use This Instead

`% v is a visa object`
`v.BytesAvailableFcnCount = 5`
`v.BytesAvailableFcnMode = "byte"`
`v.BytesAvailableFcn = @mycallback`
 	
`function mycallback(src,evt)`
`   data = fread(src,src.BytesAvailableFcnCount);`
`   disp(evt)`
`   disp(evt.Data)`
`end`
 	
`    Type: 'BytesAvailable'`
`    Data: [1×1 struct]`
 	
`    AbsTime: [2019 12 21 16 35 9.7032]`
	`% v is a visadev object`
	`configureCallback(v,"byte",5,@mycallback);`
	`function mycallback(src,evt)`
	`   data = read(src,src.BytesAvailableFcnCount);`
	`   disp(evt)`
	`end`
	
	`  ByteAvailableInfo with properties:`
	
	`    BytesAvailableFcnCount: 5`
	`                   AbsTime: 21-Dec-2019 12:23:01`

`% v is a visa object`
`v.Terminator = "CR"`
`v.BytesAvailableFcnMode = "terminator"`
`v.BytesAvailableFcn = @mycallback`
 	
`function mycallback(src,evt)`
`   data = fscanf(src);`
`   disp(evt)`
`   disp(evt.Data)`
`end`
 	
`    Type: 'BytesAvailable'`
`    Data: [1×1 struct]`
 	
`    AbsTime: [2019 12 21 16 35 9.7032]`
	`% v is a visadev object`
	`configureTerminator(v,"CR")`
	`configureCallback(v,"terminator",@mycallback)`
	
	`function mycallback(src,evt)`
	`   data = readline(src);`
	`   disp(evt)`
	`end`
	
	`  TerminatorAvailableInfo with properties:`
	
	`                   AbsTime: 21-Dec-2019 12:23:01`

For more information, see `configureCallback`.

See Also

`visadev`

More About

- R2020b VISA Interface Topics

Controlling Instruments Using the Serial Port

This chapter describes specific issues related to controlling instruments that use the serial port.

- “Serial Port Overview” on page 6-2
- “Create Serial Port Object” on page 6-13
- “Configure Serial Port Communication Settings” on page 6-15
- “Write and Read Serial Port Data” on page 6-17
- “Use Callbacks for Serial Port Communication” on page 6-21
- “Use Serial Port Control Pins” on page 6-22
- “Transition Your Code to serialport Interface” on page 6-26

Serial Port Overview

In this section...

"What Is Serial Communication?" on page 6-2
"Serial Port Interface Standard" on page 6-2
"Supported Platforms" on page 6-3
"Connecting Two Devices with a Serial Cable" on page 6-3
"Serial Port Signals and Pin Assignments" on page 6-4
"Serial Data Format" on page 6-6
"Find Serial Port Information for Your Platform" on page 6-9

What Is Serial Communication?

Serial communication is the most common low-level protocol for communicating between two or more devices. Normally, one device is a computer, while the other device can be a modem, a printer, another computer, or a scientific instrument such as an oscilloscope or a function generator.

As the name suggests, the serial port sends and receives bytes of information in a serial fashion—one bit at a time. These bytes are transmitted using either a binary format or a text (ASCII) format.

For many serial port applications, you can communicate with your instrument without detailed knowledge of how the serial port works. Communication is established through a serial port object, which you create in the MATLAB workspace.

If your application is straightforward, or if you are already familiar with the topics mentioned above, you might want to begin with "Create Serial Port Object" on page 6-13. If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to "Get Started with Instrument Control Toolbox".

Serial Port Interface Standard

Over the years, several serial port interface standards for connecting computers to peripheral devices have been developed. These standards include RS-232, RS-422, and RS-485 — all of which are supported by the `serialport` object. The most widely used standard is RS-232.

The current version of this standard is designated TIA/EIA-232C, which is published by the Telecommunications Industry Association. However, the term "RS-232" is still in popular use, and is used here to refer to a serial communication port that follows the TIA/EIA-232 standard. RS-232 defines these serial port characteristics:

- Maximum bit transfer rate and cable length
- Names, electrical characteristics, and functions of signals
- Mechanical connections and pin assignments

Primary communication uses three pins: the Transmit Data pin, the Receive Data pin, and the Ground pin. Other pins are available for data flow control, but are not required.

Note This guide assumes that you are using the RS-232 standard. Refer to your device documentation to see which interface standard you can use.

Supported Platforms

The MATLAB serial port interface is supported on:

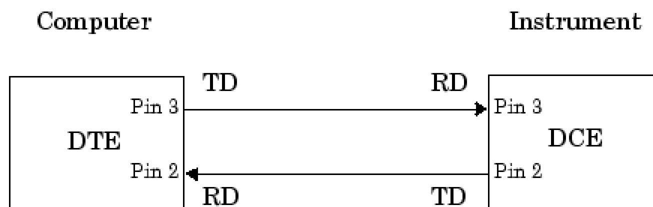
- Linux 64-bit
- macOS 64-bit
- Microsoft® Windows 64-bit

Connecting Two Devices with a Serial Cable

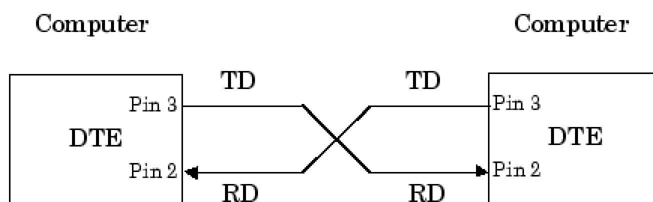
The RS-232 and RS-485 standard defines the two devices connected with a serial cable as the data terminal equipment (DTE) and data circuit-terminating equipment (DCE). This terminology reflects the RS-232 origin as a standard for communication between a computer terminal and a modem.

In this guide, your computer is considered a DTE, while peripheral devices such as modems and printers are considered DCEs. Note that many scientific instruments function as DTEs.

Because RS-232 mainly involves connecting a DTE to a DCE, the pin assignment definitions specify straight-through cabling, where pin 1 is connected to pin 1, pin 2 is connected to pin 2, and so on. A DTE-to-DCE serial connection using the transmit data (TD) pin and the receive data (RD) pin is shown below. Refer to "Serial Port Signals and Pin Assignments" on page 6-4 for more information about serial port pins.



If you connect two DTEs or two DCEs using a straight serial cable, then the TD pin on each device is connected to the other, and the RD pin on each device is connected to the other. Therefore, to connect two like devices, you must use a *null modem* cable. As shown below, null modem cables cross the transmit and receive lines in the cable.

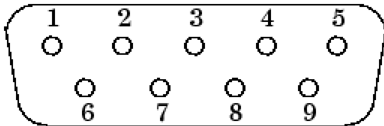


Note You can connect multiple RS-422 or RS-485 devices to a serial port. If you have an RS-232/ RS-485 adaptor, then you can use the `serialport` object with these devices.

Serial Port Signals and Pin Assignments

Serial ports consist of two signal types: *data signals* and *control signals*. To support these signal types, as well as the signal ground, the RS-232 standard defines a 25-pin connection. However, most PCs and UNIX® platforms use a 9-pin connection. In fact, only three pins are required for serial port communications: one for receiving data, one for transmitting data, and one for the signal ground.

The following figure shows a pin assignment scheme for a nine-pin male connector on a DTE.



This table describes the pins and signals associated with the nine-pin connector. Refer to the RS-232 or the RS-485 standard for a description of the signals and pin assignments for a 25-pin connector.

Serial Port Pin and Signal Assignments

Pin	Label	Signal Name	Signal Type
1	CD	Carrier Detect	Control
2	RD	Received Data	Data
3	TD	Transmitted Data	Data
4	DTR	Data Terminal Ready	Control
5	GND	Signal Ground	Ground
6	DSR	Data Set Ready	Control
7	RTS	Request to Send	Control
8	CTS	Clear to Send	Control
9	RI	Ring Indicator	Control

The term “data set” is synonymous with “modem” or “device,” while the term “data terminal” is synonymous with “computer.”

Note The serial port pin and signal assignments are with respect to the DTE. For example, data is transmitted from the TD pin of the DTE to the RD pin of the DCE.

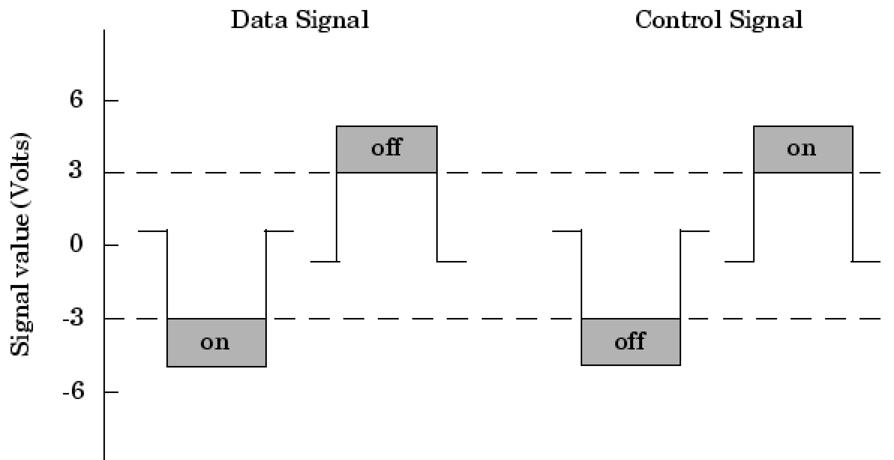
Signal States

Signals can be in either an *active state* or an *inactive state*. An active state corresponds to the binary value 1, while an inactive state corresponds to the binary value 0. An active signal state is often described as *logic 1*, *on*, *true*, or a *mark*. An inactive signal state is often described as *logic 0*, *off*, *false*, or a *space*.

For data signals, the “on” state occurs when the received signal voltage is more negative than –3 volts, while the “off” state occurs for voltages more positive than 3 volts. For control signals, the “on” state occurs when the received signal voltage is more positive than 3 volts, while the “off” state occurs for voltages more negative than –3 volts. The voltage between –3 volts and +3 volts is considered a transition region, and the signal state is undefined.

To bring the signal to the “on” state, the controlling device *unasserts* (or *lowers*) the value for data pins and *asserts* (or *raises*) the value for control pins. Conversely, to bring the signal to the “off” state, the controlling device asserts the value for data pins and unasserts the value for control pins.

The following figure depicts the “on” and “off” states for a data signal and for a control signal.



Data Pins

Most serial port devices support *full-duplex* communication, meaning that they can send and receive data at the same time. Therefore, separate pins are used for transmitting and receiving data. For these devices, the TD, RD, and GND pins are used. However, some types of serial port devices support only one-way or *half-duplex* communications. For these devices, only the TD and GND pins are used. This guide assumes that a full-duplex serial port is connected to your device.

The TD pin carries data transmitted by a DTE to a DCE. The RD pin carries data that is received by a DTE from a DCE.

Control Pins

The control pins of a nine-pin serial port are used to determine the presence of connected devices and control the flow of data. The control pins include:

- “RTS and CTS Pins” on page 6-5
- “DTR and DSR Pins” on page 6-6
- “CD and RI Pins” on page 6-6

RTS and CTS Pins

The RTS and CTS pins are used to signal whether the devices are ready to send or receive data. This type of data flow control — called hardware handshaking — is used to prevent data loss during transmission. When enabled for both the DTE and DCE, hardware handshaking using RTS and CTS follows these steps:

1. The DTE asserts the RTS pin to instruct the DCE that it is ready to receive data.
2. The DCE asserts the CTS pin, indicating that it is clear to send data over the TD pin. If data can no longer be sent, the CTS pin is unasserted.
3. The data is transmitted to the DTE over the TD pin. If data can no longer be accepted, the RTS pin is unasserted by the DTE and the data transmission is stopped.

To enable hardware handshaking, refer to “Controlling the Flow of Data: Handshaking” on page 6-24.

DTR and DSR Pins

Many devices use the DSR and DTR pins to signal if they are connected and powered. Signaling the presence of connected devices using DTR and DSR follows these steps:

1. The DTE asserts the DTR pin to request that the DCE connect to the communication line.
2. The DCE asserts the DSR pin to indicate that it is connected.
3. DCE unasserts the DSR pin when it is disconnected from the communication line.

The DTR and DSR pins were originally designed to provide an alternative method of hardware handshaking. However, the RTS and CTS pins are usually used in this way, and not the DSR and DTR pins. Refer to your device documentation to determine its specific pin behavior.

CD and RI Pins

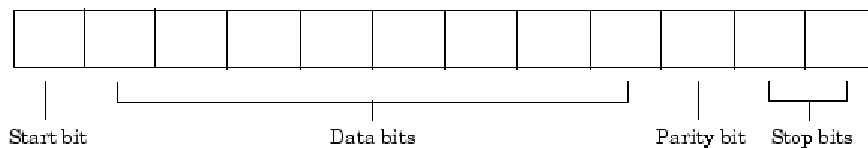
The CD and RI pins are typically used to indicate the presence of certain signals during modem-modem connections.

CD is used by a modem to signal that it has made a connection with another modem, or has detected a carrier tone. CD is asserted when the DCE is receiving a signal of a suitable frequency. CD is unasserted if the DCE is not receiving a suitable signal.

RI is used to indicate the presence of an audible ringing signal. RI is asserted when the DCE is receiving a ringing signal. RI is unasserted when the DCE is not receiving a ringing signal (for example, it is between rings).

Serial Data Format

The serial data format includes one start bit, between five and eight data bits, and one stop bit. A parity bit and an additional stop bit might be included in the format as well. This diagram illustrates the serial data format.



The format for serial port data is often expressed using the following notation:

number of data bits - parity type - number of stop bits

For example, 8-N-1 is interpreted as eight data bits, no parity bit, and one stop bit, while 7-E-2 is interpreted as seven data bits, even parity, and two stop bits.

The data bits are often referred to as a *character* because these bits usually represent an ASCII character. The remaining bits are called *framing bits* because they frame the data bits.

Bytes Versus Values

The collection of bits that compose the serial data format is called a *byte*. At first, this term might seem inaccurate because a byte is 8 bits and the serial data format can range between 7 bits and 12

bits. However, when serial data is stored on your computer, the framing bits are stripped away, and only the data bits are retained. Moreover, eight data bits are always used regardless of the number of data bits specified for transmission, with the unused bits assigned a value of 0.

When reading or writing data, you might need to specify a *value*, which can consist of one or more bytes. For example, if you read one value from a device using the `int32` format, then that value consists of four bytes. For more information about reading and writing values, refer to "Write and Read Serial Port Data" on page 6-17.

Synchronous and Asynchronous Communication

The RS-232 and the RS-485 standards support two types of communication protocols: synchronous and asynchronous.

Using the synchronous protocol, all transmitted bits are synchronized to a common clock signal. The two devices initially synchronize themselves to each other, and then continually send characters to stay synchronized. Even when actual data is not really being sent, a constant flow of bits allows each device to know where the other is at any given time. That is, each bit that is sent is either actual data or an idle character. Synchronous communications allows faster data transfer rates than asynchronous methods, because additional bits to mark the beginning and end of each data byte are not required.

Using the asynchronous protocol, each device uses its own internal clock, resulting in bytes that are transferred at arbitrary times. So, instead of using time as a way to synchronize the bits, the data format is used.

In particular, the data transmission is synchronized using the start bit of the word, while one or more stop bits indicate the end of the word. The requirement to send these additional bits causes asynchronous communications to be slightly slower than synchronous. However, it has the advantage that the processor does not have to deal with the additional idle characters. Most serial ports operate asynchronously.

Note In this guide, the terms "synchronous" and "asynchronous" refer to whether read or write operations block access to the MATLAB Command Window.

How Are the Bits Transmitted?

By definition, serial data is transmitted one bit at a time. The order in which the bits are transmitted follows these steps:

1. The start bit is transmitted with a value of 0.
2. The data bits are transmitted. The first data bit corresponds to the least significant bit (LSB), while the last data bit corresponds to the most significant bit (MSB).
3. The parity bit (if defined) is transmitted.
4. One or two stop bits are transmitted, each with a value of 1.

The number of bits transferred per second is given by the *baud rate*. The transferred bits include the start bit, the data bits, the parity bit (if defined), and the stop bits.

Start and Stop Bits

As described in "Synchronous and Asynchronous Communication" on page 6-7, most serial ports operate asynchronously. This means that the transmitted byte must be identified by start and stop

bits. The start bit indicates when the data byte is about to begin and the stop bit indicates when the data byte has been transferred. The process of identifying bytes with the serial data format follows these steps:

1. When a serial port pin is idle (not transmitting data), then it is in an "on" state.
2. When data is about to be transmitted, the serial port pin switches to an "off" state due to the start bit.
3. The serial port pin switches back to an "on" state due to the stop bit(s). This indicates the end of the byte.

Data Bits

The data bits transferred through a serial port can represent device commands, sensor readings, error messages, and so on. The data can be transferred as either binary data or as text (ASCII) data.

Most serial ports use between five and eight data bits. Binary data is typically transmitted as eight bits. Text-based data is transmitted as either seven bits or eight bits. If the data is based on the ASCII character set, then a minimum of seven bits is required because there are 2^7 or 128 distinct characters. If an eighth bit is used, it must have a value of 0. If the data is based on the extended ASCII character set, then eight bits must be used because there are 2^8 or 256 distinct characters.

Parity Bit

The parity bit provides simple error (parity) checking for the transmitted data. This table describes the types of parity checking.

Parity Types

Parity Type	Description
Even	The data bits plus the parity bit produce an even number of 1s.
Mark	The parity bit is always 1.
Odd	The data bits plus the parity bit produce an odd number of 1s.
Space	The parity bit is always 0.

Mark and space parity checking are seldom used because they offer minimal error detection. You can choose not to use parity checking at all.

The parity checking process follows these steps:

1. The transmitting device sets the parity bit to 0 or to 1 depending on the data bit values and the type of parity checking selected.
2. The receiving device checks if the parity bit is consistent with the transmitted data. If it is, then the data bits are accepted. If it is not, then an error is returned.

Note Parity checking can detect only one-bit errors. Multiple-bit errors can appear as valid data.

For example, suppose the data bits 01110001 are transmitted to your computer. If even parity is selected, then the parity bit is set to 0 by the transmitting device to produce an even number of 1s. If odd parity is selected, then the parity bit is set to 1 by the transmitting device to produce an odd number of 1s.

Find Serial Port Information for Your Platform

You can find serial port information using the resources provided by Windows and UNIX platforms.

Note Your operating system provides default values for all serial port settings. However, these settings are overridden by your MATLAB code, and have no effect on your serial port application.

You can also use the `instrhwinfo` function to return the available serial ports programmatically.

Use the serialportlist Function to Find Available Ports

The `serialportlist` function returns a list of all serial ports on a system, including virtual serial ports provided by USB-to-serial devices and Bluetooth Serial Port Profile devices. The function provides a list of the serial ports that you have access to on your computer and can use for serial port communication. For example:

```
serialportlist
```

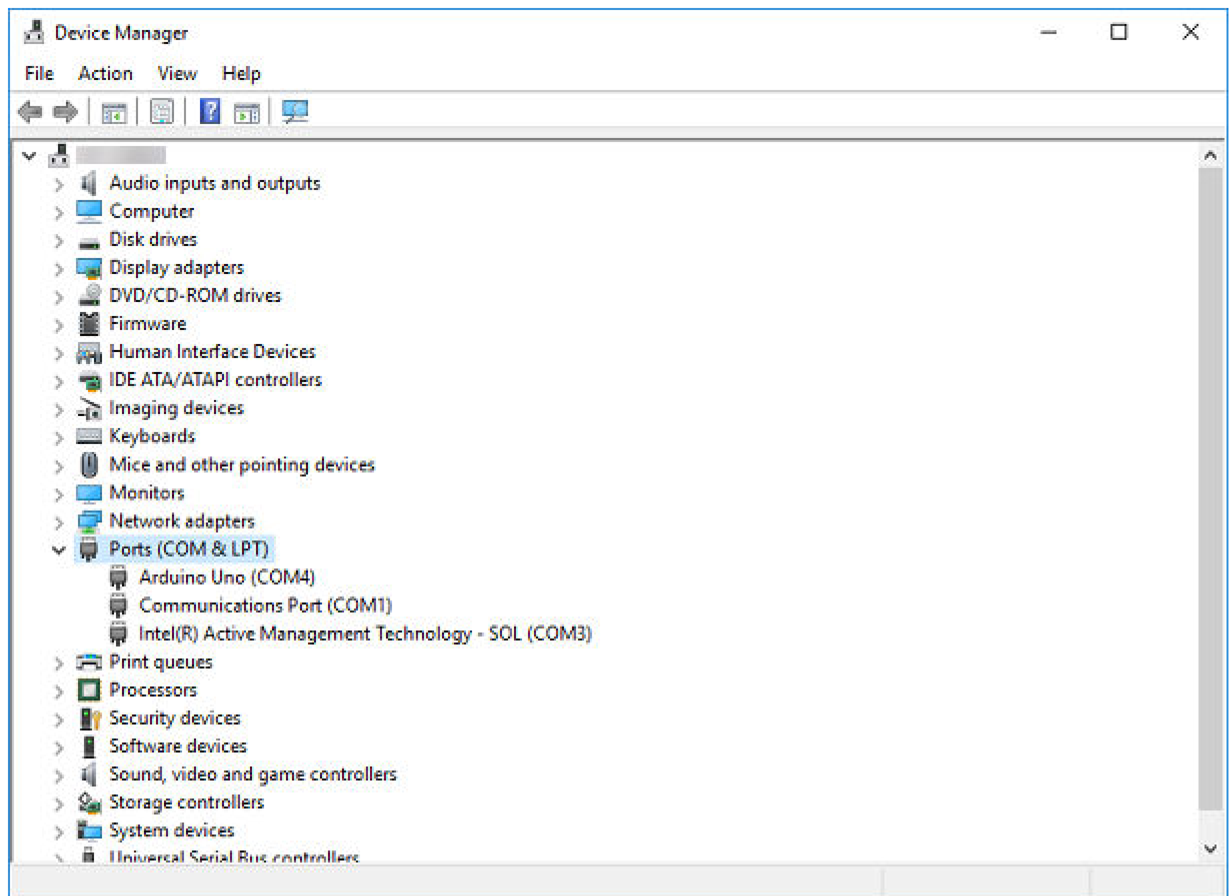
```
ans = 

  1×3 string array

    "COM1"    "COM3"    "COM4"
```

Note The `serialportlist` function shows both available and in-use ports on Windows and macOS systems, but on Linux, it shows only available ports and not in-use ports.

Windows Platform

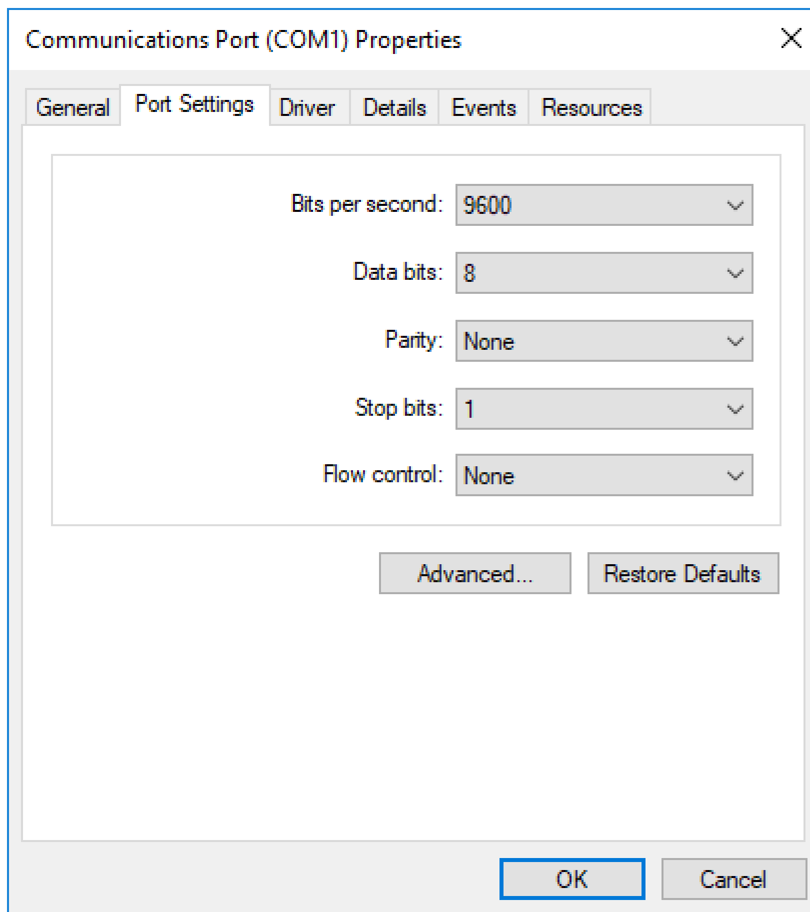
You can access serial port information through the **Device Manager**.

1. Open **Device Manager**.
2. Expand the **Ports (COM & LPT)** list.



3 Double-click the **Communications Port (COM1)** item.

4 Select the **Port Settings** tab.



UNIX Platform

To find serial port information for UNIX platforms, you need to know the serial port names. These names can vary between different operating systems.

On Linux, serial port devices are typically named `ttyS0`, `ttyS1`, and so on. You can use the `setserial` command to display or configure serial port information. For example, to display which serial ports are available:

```
setserial -bg /dev/ttyS*
```

```
/dev/ttyS0 at 0x03f8 (irq = 4) is a 16550A
/dev/ttyS1 at 0x02f8 (irq = 3) is a 16550A
```

To display detailed information about `ttyS0`:

```
setserial -ag /dev/ttyS0
```

```
/dev/ttyS0, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4
        Baud_base: 115200, close_delay: 50, divisor: 0
        closing_wait: 3000, closing_wait2: infinte
        Flags: spd_normal skip_test session_lockout
```

Note If the `setserial -ag` command does not work, make sure that you have read and write permission for the port.

For all supported UNIX platforms, including macOS, you can use the `stty` command to display or configure serial port information. For example, to display serial port properties for `ttyS0`, type:

```
stty -a < /dev/ttyS0
```

To configure the baud rate as 4800 bits per second, type:

```
stty speed 4800 < /dev/ttyS0 > /dev/ttyS0
```

Note This example shows how to set `tty` parameters, not the baud rate. To set the baud rate using the MATLAB serial interface, refer to “Configure Serial Port Communication Settings” on page 6-15.

Create Serial Port Object

In this section...
"Create a Serial Port Object" on page 6-13
"Serial Port Object Display" on page 6-14

Create a Serial Port Object

You create a serial port object with the `serialport` function. `serialport` requires the name of the serial port connected to your device and the baud rate as input arguments. You can also configure property values during object creation using name-value pair arguments.

Each serial port object is associated with one serial port. For example, connect to a device that is on serial port COM1 and configured for a baud rate of 4800.

```
s = serialport("COM1",4800);
```

If the specified port does not exist, or if it is in use, you cannot connect the serial port object to the device. The port name depends on the platform that the serial port is on.

You can also use `instrhwinfo` to see a list of available serial ports.

```
instrhwinfo("serialport")
```

You can also use the `serialportlist` function to return a list of all serial ports on a system, including virtual serial ports provided by USB-to-serial devices and Bluetooth Serial Port Profile devices. The list shows all serial ports that you have access to on your computer and can use for serial port communication.

```
serialportlist

ans = 

  1×3 string array

    "COM1"    "COM3"    "COM4"
```

This table shows an example of serial constructors on different platforms.

Platform	Serial Constructor
Linux 64-bit	`s = serialport("/dev/ttyS0",9600);`
macOS 64-bit	`s = serialport("/dev/tty.KeySerial1",9600);`
Microsoft Windows 64-bit	`s = serialport("COM1",9600);`

Note The first time you try to access a serial port in MATLAB using the `s = serialport("COM1",9600)` call, make sure that the port is free and is not already open in any other application. If the port is open in another application, MATLAB cannot access it. After you access the serial port in MATLAB, you can open the same port in other applications, and MATLAB continues to use it along with any other application that has it open as well.

Serial Port Object Display

The serial port object provides a convenient display that summarizes important configuration and state information. You can invoke the display summary in three ways:

- Type the serial port object variable name at the command line.
- Exclude the semicolon when creating a serial port object.
- Exclude the semicolon when configuring properties using dot notation.

You can also display summary information using the workspace browser by right-clicking an instrument object and selecting **Display Summary** from the context menu.

The display summary for the serial port object `s` on a Windows machine is given here.

```
s = serialport("COM4",9600)
```

```
s = 

  Serialport with properties:

                 Port: "COM4"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, all methods

                      Port: "COM4"
                  BaudRate: 9600
         NumBytesAvailable: 0

                 ByteOrder: "little-endian"
                  DataBits: 8
                  StopBits: 1
                    Parity: "none"
               FlowControl: "none"
                   Timeout: 10
                Terminator: "LF"

     BytesAvailableFcnMode: "off"
    BytesAvailableFcnCount: 64
         BytesAvailableFcn: []
           NumBytesWritten: 0

          ErrorOccurredFcn: []
                  UserData: []
```

Use dot notation to configure and display property values.

```
s.BaudRate = 4800;
s.BaudRate
```

```
ans =

        4800
```

For more information about configuring these properties, see `serialport`.

Configure Serial Port Communication Settings

Before you can write or read data, both the serial port object and the instrument must have identical communication settings. Configuring serial port communications involves specifying values for properties that control the baud rate and the "Serial Data Format" on page 6-6. These properties are as follows.

Serial Port Communication Properties

Property Name	Description
`BaudRate` on page 24-0	Specify the rate at which bits are transmitted.
`Parity` on page 24-0	Specify the type of parity checking.
`DataBits` on page 24-0	Specify the number of data bits to transmit.
`StopBits` on page 24-0	Specify the number of bits used to indicate the end of a byte.
`Terminator` on page 24-0	Specify the terminator character.

Caution If the serial port object and the instrument communication settings are not identical, you might not be able to successfully read or write data.

Refer to your instrument documentation for an explanation of its supported communication settings.

You can display the communication property values for the serial port object `s` created in "Create Serial Port Object" on page 6-13.

```
s = serialport("COM4",9600)
```

```
s = 

  Serialport with properties:

                 Port: "COM4"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, all methods

                      Port: "COM4"
                  BaudRate: 9600
         NumBytesAvailable: 0

                 ByteOrder: "little-endian"
                  DataBits: 8
                  StopBits: 1
                    Parity: "none"
               FlowControl: "none"
                   Timeout: 10
                Terminator: "LF"
```

```
  BytesAvailableFcnMode: "off"
 BytesAvailableFcnCount: 64
      BytesAvailableFcn: []
         NumBytesWritten: 0

       ErrorOccurredFcn: []
               UserData: []
```

Write and Read Serial Port Data

In this section...
“Rules for Completing Write and Read Operations” on page 6-17
“Writing and Reading Text Data” on page 6-17
“Writing and Reading Binary Data” on page 6-19

Rules for Completing Write and Read Operations

Completing Write Operations

A write operation using `write`, `writeline`, or `writebinblock` completes when one of these conditions is satisfied:

- The specified data is written.
- The time specified by the `Timeout` property passes.

A text command is processed by the instrument only when it receives the required terminator. For serial port objects, each occurrence of `\n` in the ASCII command is replaced with the `Terminator` property value. The default value of `Terminator` is the line feed character. Refer to the documentation for your instrument to determine the terminator required by your instrument.

Completing Read Operations

A read operation with `read`, `readline`, or `readbinblock` completes when one of these conditions is satisfied:

- The specified number of values is read.
- The time specified by the `Timeout` property passes.
- The terminator specified by the `Terminator` property is read.

Writing and Reading Text Data

This example illustrates how to communicate with a serial port instrument by writing and reading text data.

The instrument is a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1. Therefore, many of the commands in the example are specific to this instrument. A sine wave is input into channel 2 of the oscilloscope, and you want to measure the peak-to-peak voltage of the input signal.

These functions and properties are used when reading and writing text.

Function	Purpose
`readline`	Read text data from the instrument.
`writeline`	Write text data to the instrument.
“Terminator” on page 24-0	Character used to terminate commands sent to the instrument.

Note This example is Windows specific.

1 **Create a serial port object** — Create the serial port object `s` associated with the serial port COM1.

```
s = serialport("COM1",9600);
```

2 **Write and read data** — Write the `*IDN?` command to the instrument using `writeline`, and then read back the result of the command using `readline`.

```
writeline(s,"*IDN?")
s.NumBytesAvailable

ans =

     56

idn = readline(s)

idn =

     "TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04"
```

You need to determine the measurement source. Possible measurement sources include channel 1 and channel 2 of the oscilloscope.

```
writeline(s,"MEASUREMENT:IMMED:SOURCE?")
source = readline(s)

source =

     "CH1"
```

The scope is configured to return a measurement from channel 1. Because the input signal is connected to channel 2, you must configure the instrument to return a measurement from this channel.

```
writeline(s,"MEASUREMENT:IMMED:SOURCE CH2")
writeline(s,"MEASUREMENT:IMMED:SOURCE?")
source = readline(s)

source =

     "CH2"
```

You can now configure the scope to return the peak-to-peak voltage, and then request the value of this measurement.

```
writeline(s,"MEASUREMENT:MEAS1:TYPE PK2PK")
writeline(s,"MEASUREMENT:MEAS1:VALUE?")
```

Read back the result using the `readline` function.

```
ptop = readline(s)

ptop =

     "2.0199999809E0"
```

3 **Disconnect and clean up** — Clear the serial port object `s` from the MATLAB workspace when you are done working with it.

```
clear s
```

Writing and Reading Binary Data

This example explores binary read and write operations with a serial port object. The instrument used is a Tektronix® TDS 210 oscilloscope.

Functions and Properties

These functions are used when reading and writing binary data.

Function	Purpose
`read`	Read binary data from the instrument.
`write`	Write binary data to the instrument.

Configure and Connect to the Serial Object

You need to create a serial object. In this example, create a serial port object associated with the COM1 port.

```
s = serialport("COM1",9600);
```

Write Binary Data

You use the `write` function to write binary data to the instrument. A binary write operation completes when one of these conditions is satisfied:

- All the data is written.
- A timeout occurs as specified by the `Timeout` property.

Note When you perform a write operation, think of the transmitted data in terms of values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

Writing Int16 Binary Data

Write a waveform as an `int16` array.

```
write(s,"Data:Destination RefB","string");
write(s,"Data:Encdg SRPbinary","string");
write(s,"Data:Width 2","string");
write(s,"Data:Start 1","string");

t = (0:499) .* 8 * pi / 500;
data = round(sin(t) * 90 + 127);
write(s,"CURVE #3500","string");
```

Note that one `int16` value consists of two bytes. Therefore, the following command writes 1000 bytes.

```
write(s,data,"int16")
```

Reading Binary Data

You use the `read` function to read binary data from the instrument. A binary read operation completes when one of these conditions is satisfied:

- A timeout occurs as specified by the `Timeout` property.
- The specified number of values is read.

Note When you perform a read operation, think of the received data in terms of values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

Reading int16 Binary Data

Read the same waveform on channel 1 as an `int16` array.

```
read(s,"Data:Source CH1","string");
read(s,"Data:Encdg SRPbinary","string");
read(s,"Data:Width 2","string");
read(s,"Data:Start 1","string");
read(s,"Data:Stop 2500","string");
read(s,"Curve?","string")
```

Note that one `int16` value consists of two bytes. Therefore, the following command reads 512 bytes.

```
data = read(s,256,"int16");
```

Disconnect and Clean Up

If you are finished with the serial port object, clear the object from the workspace.

```
clear s
```

Use Callbacks for Serial Port Communication

In this section...
"Callback Properties" on page 6-21
"Using Callbacks" on page 6-21

Callback Properties

The properties and functions associated with callbacks are as follows.

Property or Function	Purpose
NumBytesAvailable	Number of bytes available to read
BytesAvailableFcn	Bytes available callback function
BytesAvailableFcnCount	Number of bytes of data to trigger callback
BytesAvailableFcnMode	Bytes available callback trigger mode
`configureCallback`	Set serial port callback function and trigger

Using Callbacks

This example uses a loopback device with the callback function `readSerialData` to return data to the command line when a terminator is read.

Note This example is Windows specific.

1 **Create the callback function** — Define a callback function `readSerialData` that performs a terminated string read and returns the data.

```
function readSerialData(src,~)
    data = readline(src);
    disp(data);
end
```

2 **Create an instrument object** — Create the serial port object `s` associated with serial port COM1.

```
s = serialport("COM1",9600);
```

3 **Configure properties** — Configure `s` to execute the callback function `readSerialData` when the terminator is read.

```
configureCallback(s,"terminator",@readSerialData)
```

4 **Disconnect and clean up** — Clear the objects from the MATLAB workspace when you are done.

```
clear s
```

Use Serial Port Control Pins

In this section...
“Control Pins” on page 6-22
“Signaling the Presence of Connected Devices” on page 6-22
“Controlling the Flow of Data: Handshaking” on page 6-24

Control Pins

As described in “Serial Port Signals and Pin Assignments” on page 6-4, nine-pin serial ports include six control pins. The functions and properties associated with the serial port control pins are as follows.

Function	Purpose
`getpinstatus`	Get serial pin status.
`setRTS`	Specify the state of the RTS pin.
`setDTR`	Specify the state of the DTR pin.
FlowControl	Specify the data flow control method to use.

Signaling the Presence of Connected Devices

DTEs and DCEs often use the CD, DSR, RI, and DTR pins to indicate whether a connection is established between serial port devices. Once the connection is established, you can begin to write or read data.

You can monitor the state of the CD, DSR, and RI pins with the `getpinstatus` function. You can specify the state of the DTR pin with the `setDTR` function.

The following example illustrates how these pins are used when two modems are connected to each other.

Connect Two Modems

This example (shown on a Windows machine) connects two modems to each other through the same computer, and illustrates how you can monitor the communication status for the computer-modem connections, and for the modem-modem connection. The first modem is connected to COM1, while the second modem is connected to COM2.

1 **Connect to the instruments** — After the modems are powered on, the serial port object `s1` is created for the first modem, and the serial port object `s2` is created for the second modem. both modems are configured for a baud rate of 9600 bits per second.

```
s1 = serialport("COM1",9600);
s2 = serialport("COM2",9600);
```

You can verify that the modems (data sets) are ready to communicate with the computer by examining the value of the Data Set Ready pin using the `getpinstatus` function.

```
getpinstatus(s)
```

```
ans = 

  struct with fields:

      ClearToSend: 1
     DataSetReady: 1
    CarrierDetect: 0
    RingIndicator: 0
```

The value of the `DataSetReady` field is `1`, or `true`, because both modems were powered on before they were connected to the objects.

2. **Configure properties —** Both modems are configured for a carriage return (CR) terminator using the `configureTerminator` function.

```
configureTerminator(s1,"CR")
configureTerminator(s2,"CR")
```

3. **Write and read data** — Write the `atd` command to the first modem using the `writeline` function. This command puts the modem “off the hook,” and is equivalent to manually lifting a phone receiver.

```
writeline(s1,'atd')
```

Write the `ata` command to the second modem using the `writeline` function. This command puts the modem in “answer mode,” which forces it to connect to the first modem.

```
writeline(s2,'ata')
```

After the two modems negotiate their connection, you can verify the connection status by examining the value of the Carrier Detect pin using the `getpinstatus` function.

```
getpinstatus(s)
```

```
ans = 

  struct with fields:

      ClearToSend: 1
     DataSetReady: 1
    CarrierDetect: 1
    RingIndicator: 0
```

You can also verify the modem-modem connection by reading the descriptive message returned by the second modem.

```
s2.NumBytesAvailable
```

```
ans =

     25
```

```
out = read(s2,25,"uint32")
```

```
out = 
ata
CONNECT 2400/NONE
```

Now break the connection between the two modems by using the `setDTR` function. You can verify that the modems are disconnected by examining the Carrier Detect pin value using the `getpinstatus` function.

```
setDTR(s1,false)
getpinstatus(s1)

ans = 

  struct with fields:

      ClearToSend: 1
     DataSetReady: 1
    CarrierDetect: 0
    RingIndicator: 0
```

4 **Disconnect and clean up** — Clear the objects from the MATLAB workspace when you are done.

```
clear s1 s2
```

Controlling the Flow of Data: Handshaking

Data flow control or *handshaking* is a method used for communicating between a DCE and a DTE to prevent data loss during transmission. For example, suppose your computer can receive only a limited amount of data before it must be processed. As this limit is reached, a handshaking signal is transmitted to the DCE to stop sending data. When the computer can accept more data, another handshaking signal is transmitted to the DCE to resume sending data.

If supported by your device, you can control data flow using one of these methods:

- “Hardware Handshaking” on page 6-24
- “Software Handshaking” on page 6-25

Note Although you might be able to configure your device for both hardware handshaking and software handshaking at the same time, MATLAB does not support this behavior.

You can specify the data flow control method with the `FlowControl` property. If `FlowControl` is `hardware`, then hardware handshaking is used to control data flow. If `FlowControl` is `software`, then software handshaking is used to control data flow. If `FlowControl` is `none`, then no handshaking is used.

Hardware Handshaking

Hardware handshaking uses specific serial port pins to control data flow. In most cases, these are the RTS and CTS pins. Hardware handshaking using these pins is described in “RTS and CTS Pins” on page 6-5.

If `FlowControl` is `hardware`, then the RTS and CTS pins are automatically managed by the DTE and DCE. You can return the CTS pin value with the `getpinstatus` function. You can configure the RTS pin value with the `setRTS` function.

Note Some devices also use the DTR and DSR pins for handshaking. However, these pins are typically used to indicate that the system is ready for communication, and are not used to control data transmission. In MATLAB, hardware handshaking always uses the RTS and CTS pins.

If your device does not use hardware handshaking in the standard way, then you might need to manually configure the RTS pin using the `setRTS` function. In this case, configure `FlowControl` to

`none`. If `FlowControl` is `hardware`, then the RTS value that you specify might not be honored. Refer to the device documentation to determine its specific pin behavior.

Software Handshaking

Software handshaking uses specific ASCII characters to control data flow. The following table describes these characters, known as Xon and Xoff (or XON and XOFF).

Software Handshaking Characters

Character	Integer Value	Description
Xon	17	Resume data transmission.
Xoff	19	Pause data transmission.

When you use software handshaking, the control characters are sent over the transmission line the same way as regular data. Therefore, you need only the TD, RD, and GND pins.

The main disadvantage of software handshaking is that you cannot write the Xon or Xoff characters while numerical data is being written to the instrument. This is because numerical data might contain a 17 or 19, which makes it impossible to distinguish between the control characters and the data. However, you can write Xon or Xoff while data is being asynchronously read from the instrument because you are using both the TD and RD pins.

Using Software Handshaking

Suppose you want to use software flow control in conjunction with your serial port application. To do this, you must configure the instrument and the serial port object for software flow control. For a serial port object `s` connected to a Tektronix TDS 210 oscilloscope, this configuration is accomplished with the following commands.

```
writeline(s,"RS232:SOFTF ON")
s.FlowControl = "software";
```

To pause data transfer, you write the numerical value `19` (Xoff) to the instrument.

```
write(s,19,"uint32");
```

To resume data transfer, you write the numerical value `17` (Xon) to the instrument.

```
write(s,17,"uint32");
```

Transition Your Code to serialport Interface

The `serial` function, its object functions, and its properties will be removed. Use `serialport` instead.

`serial` Interface	`serialport` Interface	Example
`seriallist`	`serialportlist`	"Discover Serial Port Devices" on page 6-26
`serial`	`serialport`	"Connect to Serial Port Device" on page 6-26
`fwrite` and `fread`	`write` and `read`	"Read and Write" on page 6-27
`fprintf`	`writeline`	"Send a Command" on page 6-27
`fscanf`, `fgetl`, and `fgets`	`readline`	"Read a Terminated String" on page 6-28
`binblockwrite`	`writebinblock`	"Write Data with the Binary Block Protocol" on page 6-29
`binblockread`	`readbinblock`	"Read Data with the Binary Block Protocol" on page 6-29
`flushinput` and `flushoutput`	`flush`	"Flush Data from Memory" on page 6-29
`Terminator`	`configureTerminator`	"Set Terminator" on page 6-30
`BytesAvailableFcnCount`, `BytesAvailableFcnMode`, `BytesAvailableFcn`, and `BytesAvailable`	`configureCallback`	"Set Up a Callback Function" on page 6-30
`PinStatus`	`getpinstatus`	"Read Serial Pin Status" on page 6-31
`DataTerminalReady` and `RequestToSend`	`setDTR` and `setRTS`	"Set Serial DTR and RTS Pin States" on page 6-31

Discover Serial Port Devices

`seriallist` is not recommended. Use `serialportlist` instead.

Connect to Serial Port Device

This example shows how to connect to a serial port device using the recommended functionality.

Functionality	Use This Instead

`s = serial("COM1");`
`s.BaudRate = 115200;`
`fopen(s)`
	`s = serialport("COM1",115200);`

For more information, see `serialport`.

Read and Write

These examples use a loopback device to show how to perform a binary write and read, write a nonterminated command string, and read a fixed-length response string using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td>% s is a serial object
fwrite(s,1:5,"uint32")
data = fread(s,5,"uint32")

data =

1
2
3
4
5</td><td>% s is a serialport object
write(s,1:5,"uint32")
data = read(s,5,"uint32")

data =

1 2 3 4 5</td></tr>
<tr><td rowspan="2">% s is a serial object
command = "start";
fwrite(s,command,"char")</td><td>% s is a serialport object
command = "start";
write(s,command,"char")</td></tr>
<tr><td>% s is a serialport object
command = "start";
write(s,command,"string")</td></tr>
<tr><td>% s is a serial object
length = 5;
resp = fread(s,length,"char")

resp =

115
116
97
114
116

resp = char(resp)'

resp =

'start'</td><td>% s is a serialport object
length = 5;
resp = read(s,length,"string")

resp =

"start"</td></tr>
</table>

For more information, see `write` or `read`.

Send a Command

This example shows how to write a terminated SCPI command using the recommended functionality.

Functionality	Use This Instead

`% s is a serial object`
`s.Terminator = "CR/LF"`
`channel = 1;`
`level = 3.44;`
`fprintf(s,"TRIGGER%d:LEVEL2 %1.2f",[channel,level]);`
	`% s is a serialport object`
	`configureTerminator(s,"CR/LF")`
	`channel = 1;`
	`level = 3.44;`
	`cmd = sprintf("TRIGGER%d:LEVEL2 %1.2f",[channel,level`
	`writeline(s,cmd)`
	
	`writeline` automatically appends the write terminator.

For more information, see `configureTerminator` or `writeline`.

Read a Terminated String

This example shows how to perform a terminated string read using the recommended functionality.

Functionality	Use This Instead

`% s is a serial object`
`fprintf(s,"MEASUREMENT:IMMED:TYPE PK2PK")`
`a = fscanf(s,"%e",6)`
 	
`a =`
 	
`    2.0200`
 	
For the format specifier `"%e"`, `fscanf` returns the terminator and the user must remove it from the string.
	`% s is a serialport object`
	`writeline(s,"MEASUREMENT:IMMED:TYPE PK2PK")`
	`a = readline(s)`
	
	`a =`
	
	`    "2.0200"`
	
	`sscanf(a,"%e")`
	
	`a =`
	
	`    2.0200`

`% s is a serial object`
`fprintf(s,"*IDN?")`
`a = fgetl(s)`
 	
`a =`
 	
`    'TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16'`
 	
`fgetl` reads until the specified terminator is reached and then discards the terminator.
	`% s is a serialport object`
	`writeline(s,"*IDN?")`
	`a = readline(s)`
	
	`a =`
	
	`    "TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CM`
	
	`readline` reads until the specified terminator is reached and then discards the terminator. There is no option to include the terminator.

Functionality	Use This Instead

`% s is a serial object`
`fprintf(s,"*IDN?")`
`a = fgets(s)`
 	
`a =`
 	
`'TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04`
`'`
 	
`fgets` reads until the specified terminator is reached and then returns the terminator.	

For more information, see `readline`.

Write Data with the Binary Block Protocol

This example shows how to write data with the IEEE standard binary block protocol using the recommended functionality.

Functionality	Use This Instead

`% s is a serial object`
`waveform = sin(2*pi*60*0:.16:60);`
`fprintf(s,"WLIS:WAV:DATA")`
`binblockwrite(s,waveform,"double")`
	`% s is a serialport object`
	`waveform = sin(2*pi*60*0:.16:60);`
	`writeline(s,"WLIS:WAV:DATA")`
	`writebinblock(s,waveform,"double")`

For more information, see `writebinblock`.

Read Data with the Binary Block Protocol

This example uses a loopback device to show how to read data with the IEEE standard binary block protocol using the recommended functionality.

Functionality	Use This Instead

`% s is a serial object`
`fprintf(s,"CURVe?")`
`data = binblockread(s,"double")`
 	
`data =`
 	
`1`
`2`
`3`
`4`
`5`
	`% s is a serialport object`
	`writeline(s,"CURVe?")`
	`data = readbinblock(s,"double")`
	
	`data =`
	
	`1     2     3     4     5`

For more information, see `readbinblock`.

Flush Data from Memory

This example shows how to flush data from the buffer using the recommended functionality.

Functionality	Use This Instead
`% s is a serial object` `flushinput(s)`	`% s is a serialport object` `flush(s,"input")`
`% s is a serial object` `flushoutput(s)`	`% s is a serialport object` `flush(s,"output")`

`% s is a serial object`
`flushinput(s)`
`flushoutput(s)`
	`% s is a serialport object`
	`flush(s)`

For more information, see `flush`.

Set Terminator

This example shows how to set the terminator using the recommended functionality.

Functionality	Use This Instead
`% s is a serial object` `s.Terminator = "CR/LF";`	`% s is a serialport object` `configureTerminator(s,"CR/LF")`
`% s is a serial object` `s.Terminator = {"CR/LF" [10]};`	`% s is a serialport object` `configureTerminator(s,"CR/LF",10)`

For more information, see `configureTerminator`.

Set Up a Callback Function

This example uses a loopback device to show how to set up a callback function using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>
s = serial("COM5","BaudRate",115200)
s.BytesAvailableFcnCount = 5
s.BytesAvailableFcnMode = "byte"
s.BytesAvailableFcn = @instrcallback

fopen(s)

function instrcallback(src,evt)
   data = fread(src,src.BytesAvailable)
   disp(evt)
   disp(evt.Data)
end

data =

     1
     2
     3
     4
     5

    Type: 'BytesAvailable'
    Data: [1×1 struct]

    AbsTime: [2019 5 2 16 35 9.6710]
</pre></td><td><pre>
s = serialport("COM5",115200)
configureCallback(s,"byte",5,@instrcallback);

function instrcallback(src,evt)
   data = read(src,src.NumBytesAvailable,"uint8")
   disp(evt)
end

data =

     1   2   3   4   5

  DataAvailableInfo with properties:

    BytesAvailableFcnCount: 5
                   AbsTime: 02-May-2019 15:54:09
</pre></td></tr>
</table>

For more information, see `configureCallback`.

Read Serial Pin Status

This example shows how to read serial pin status using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>
% s is a serial object
s.PinStatus

ans =

  struct with fields:

    CarrierDetect: 'on'
      ClearToSend: 'on'
     DataSetReady: 'on'
    RingIndicator: 'on'
</pre></td><td><pre>
% s is a serialport object
status = getpinstatus(s)

status =

  struct with fields:

      ClearToSend: 1
     DataSetReady: 1
    CarrierDetect: 1
    RingIndicator: 1
</pre></td></tr>
</table>

For more information, see `getpinstatus`.

Set Serial DTR and RTS Pin States

This example shows how to set serial DTR and RTS pin states using the recommended functionality.

Functionality	Use This Instead
`% s is a serial object` `s.DataTerminalReady = "on";`	`% s is a serialport object` `setDTR(s,true)`
`% s is a serial object` `s.RequestToSend = "off";`	`% s is a serialport object` `setRTS(s,false)`

For more information, see `setDTR` or `setRTS`.

See Also

`serialportlist` | `serialport`

Controlling Instruments Using TCP/IP

This chapter describes specific features related to controlling instruments that use the TCP/IP protocols.

- “TCP/IP Communication Overview” on page 7-2
- “Create TCP/IP Client and Configure Settings” on page 7-3
- “Write and Read Data over TCP/IP Interface” on page 7-7
- “Use Callbacks for TCP/IP Communication” on page 7-10
- “Configure Connection in TCP/IP Explorer” on page 7-11
- “Events and Callbacks” on page 7-12
- “Rules for Completing Read and Write Operations over TCP/IP and UDP” on page 7-15
- “Basic Workflow to Read and Write Data over TCP/IP” on page 7-17
- “Read and Write ASCII Data over TCP/IP” on page 7-19
- “Read and Write Binary Data over TCP/IP” on page 7-23
- “Asynchronous Read and Write Operations over TCP/IP” on page 7-27
- “TCP/IP and UDP Comparison” on page 7-32
- “Communicate Using TCP/IP Server Sockets” on page 7-34
- “Transition Your Code to tcpclient Interface” on page 7-36
- “Transition Your Code to tcpserver Interface” on page 7-42

TCP/IP Communication Overview

Transmission Control Protocol (TCP) is a transport protocol layered on top of the Internet Protocol (IP) and is one of the most used networking protocols. The MATLAB TCP/IP client support uses raw socket communication and lets you connect to remote hosts from MATLAB for reading and writing data. For example, you can connect to a remote weather station, acquire data, and plot the data.

- **Connection-based protocol** — The two ends of the communication link must be connected at all times during the communication.
- **Streaming protocol** — TCP/IP has a long stream of data that is transmitted from one end of the connection to the other end, and another long stream of data flowing in the opposite direction. The TCP/IP stack at one end is responsible for breaking the stream of data into packets and sending those packets, while the stack at the other end is responsible for reassembling the packets into a data stream using information in the packet headers.
- **Reliable protocol** — The packets sent by TCP/IP contain a unique sequence number. The starting sequence number is communicated to the other side at the beginning of communication. The receiver acknowledges each packet, and the acknowledgment contains the sequence number so that the sender knows which packet was acknowledged. Because the sender gets an acknowledgement for each packet received, the sender knows when packets do not arrive and can retransmit them. Also, packets that arrive out of sequence can be reassembled in the proper order by the receiver.

  Timeouts can be established because the sender knows (from the first few packets) how long it takes on average for a packet to be sent and its acknowledgment received.

You can create a TCP/IP connection to a server or hardware and perform read/write operations. Use the `tcpclient` function to create the connection, and the `write` and `read` functions for synchronously reading and writing data.

See Also

`tcpclient` | `tcpserver`

More About

- “Create TCP/IP Client and Configure Settings” on page 7-3
- “Write and Read Data over TCP/IP Interface” on page 7-7

Create TCP/IP Client and Configure Settings

MATLAB TCP/IP client support lets you connect to remote hosts or hardware from MATLAB for reading and writing data. The typical workflow is:

- Create a TCP/IP connection to a server or hardware.
- Configure the connection if necessary.
- Perform read and write operations.
- Clear and close the connection.

To communicate over the TCP/IP interface, first create a `tcpclient` object.

```
t = tcpclient(address,port);
```

The address can be either a remote host name or a remote IP address. In both cases, the port must be a positive integer between 1 and 65535.

Create Object Using Host Name

Create the TCP/IP object `t` using the host address shown and port `80`.

```
t = tcpclient("www.mathworks.com",80)
```

```
t = 
  tcpclient with properties:

              Address: 'www.mathworks.com'
                 Port: 80
    NumBytesAvailable: 0

  Show all properties, functions
```

When you connect using a host name, such as a specified web address or `'localhost'`, the IP address defaults to IPv6 format. If the server you are connecting to is expecting IPv4 format, connection fails. For IPv4, you can create a connection by specifying an explicit IP address rather than a host name.

Create Object Using IP Address

Create the TCP/IP object `t` using the IP address shown and port `80`.

```
t = tcpclient("144.212.130.17",80)
```

```
t = 
  tcpclient with properties:

              Address: '144.212.130.17'
                 Port: 80
    NumBytesAvailable: 0

  Show all properties, functions
```

Set Timeout Property

Create the object and use a name-value pair argument to set the `Timeout` value. The `Timeout` parameter specifies the waiting time to complete read and write operations in seconds, and the default value is `10`. You can change the value either during object creation or after you create the object.

Create a TCP/IP object with a timeout of `20` seconds.

```
t = tcpclient("144.212.130.17",80,"Timeout",20)
```

```
t =
  tcpclient with properties:

              Address: '144.212.130.17'
                 Port: 80
    NumBytesAvailable: 0

  Show all properties, functions
```

View the `Timeout` property.

```
t.Timeout
```

```
ans =

    20
```

The output reflects the `Timeout` property change.

Set Connect Timeout Property

Create the object and use a name-value pair argument to set the `ConnectTimeout` value. The `ConnectTimeout` parameter specifies the maximum time in seconds to wait for a connection request to the specified remote host to succeed or fail. The value must be greater than or equal to 1. If you do not specify `ConnectTimeout`, it has the default value of `Inf`. You can specify the value only during object creation.

Create a TCP/IP object and specify `ConnectTimeout` as `10` seconds.

```
t = tcpclient("144.212.130.17",80,"ConnectTimeout",10)
```

```
t =
  tcpclient with properties:

              Address: '144.212.130.17'
                 Port: 80
    NumBytesAvailable: 0

  Show all properties, functions
```

View the `ConnectTimeout` property.

```
t.ConnectTimeout
```

```
ans =

    10
```

The output reflects the `ConnectTimeout` property change.

Note If you specify an invalid address or port or the connection to the server cannot be established, the object is not created.

View TCP/IP Object Properties

After you create a `tcpclient` object, you can view a full list of properties and their values. Click `properties` in the `tcpclient` output.

```
t = tcpclient("www.mathworks.com",80)
```

```
t =
  tcpclient with properties:

              Address: 'www.mathworks.com'
                 Port: 80
    NumBytesAvailable: 0

  Show all properties, functions

                   Address: 'www.mathworks.com'
                      Port: 80
         NumBytesAvailable: 0

            ConnectTimeout: Inf
                   Timeout: 10
                 ByteOrder: "little-endian"
                Terminator: "LF"

     BytesAvailableFcnMode: "off"
    BytesAvailableFcnCount: 64
         BytesAvailableFcn: []
           NumBytesWritten: 0

          ErrorOccurredFcn: []
                  UserData: []
```

For more information about how to configure these properties, see “Properties” on page 24-348.

You can use the `configureTerminator` and `configureCallback` functions to configure certain properties.

See Also

`tcpclient`

More About

- “Write and Read Data over TCP/IP Interface” on page 7-7

Write and Read Data over TCP/IP Interface

In this section...
“Write Data” on page 7-7
“Read Data” on page 7-7
“Acquire Data from Weather Station Server” on page 7-8
“Read Page from Website” on page 7-8

Write Data

The `write` function synchronously writes data to the remote host connected to the `tcpclient` object. First specify the data, then write the data. The function waits until the specified number of values is written to the remote host.

In this example, a `tcpclient` object `t` already exists.

```
% Create a variable called data
data = 1:10;

% Write the data to the object t
write(t, data)
```

Note For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to the data type you set if you specified another data type.

Read Data

The `read` function synchronously reads data from the remote host connected to the `tcpclient` object and returns the data. There are three read options:

- Read all bytes available (no arguments).
- Optionally specify the number of bytes to read.
- Optionally specify the data type.

If you do not specify a size, the default read uses the `BytesAvailable` property value, which is equal to the number of bytes available in the input buffer.

In these examples, a `tcpclient` object `t` already exists.

```
% Read all bytes available.
read(t)

% Specify the number of bytes to read, 5 in this case.
read(t,5)

% Specify the number of bytes to read, 10, and the data type, double.
read(t,10,"double")
```

Note For any read or write operation, the data type is converted to `uint8` for the data transfer. It is then converted back to the data type you set if you specified another data type.

Acquire Data from Weather Station Server

One of the primary uses of TCP/IP communication is to acquire data from a server. This example shows how to acquire and plot data from a remote weather station.

Note The IP address in this example is not a working IP address. The example shows how to connect to a remote server. Substitute the address shown here with the IP address or host name of a server you want to communicate with.

---

1 Create the `tcpclient` object using the address shown here and port `1045`.

```
t = tcpclient("172.28.154.231",1045)
```

```
t = 
  tcpclient with properties:

              Address: '172.28.154.231'
                 Port: 1045
    NumBytesAvailable: 0

  Show all properties, functions
```

2 Acquire data using the `read` function. Specify the number of bytes to read as `30`, for 10 samples from three sensors (temperature, pressure, and humidity). Specify the data type as `double`.

```
data = read(t,30,"double");
```

3 Reshape the 1-by-30 data into 10-by-3 data to show one column each for temperature, pressure, and humidity.

```
data = reshape(data,[3,10]);
```

4 Plot the temperature.

```
subplot(311);
plot(data(:,1));
```

5 Plot the pressure.

```
subplot(312);
plot(data(:,2));
```

6 Plot the humidity.

```
subplot(313);
plot(data(:,3));
```

7 Close the connection between the TCP/IP client object and the remote host by clearing the object.

```
clear t
```

Read Page from Website

In this example, you read a page from the RFC Editor Web site using a TCP/IP object.

1 Create a TCP/IP object. Port `80` is the standard port for web servers.

```
t = tcpclient("www.rfc-editor.org",80);
```

Set the `Terminator` property of the TCP/IP object.

```
configureTerminator(t,"LF","CR/LF");
```

2 You can now communicate with the server using the `writeline` and `readline` functions.

   To ask a web server to send a web page, use the `GET` command. You can ask for a text file from the RFC Editor Web site using `'GET` *`(path/filename)`*`'`.

```
writeline(t,"GET /rfc/rfc793.txt");
```

   The server receives the command and sends back the web page. You can see if any data was sent back by looking at the `NumBytesAvailable` property of the object.

```
t.NumBytesAvailable
```

   Now you can start to read the web page data. By default, `readline` reads one line at a time. You can read lines of data until the `NumBytesAvailable` value is 0. Note that you do not see a rendered web page; the HTML file data scrolls by on the screen.

```
while (t.NumBytesAvailable > 0)
    A = readline(t)
end
```

3 If you want to do more communication, you can continue to read and write data. If you are done with the object, clear it.

```
clear t
```

See Also

`read` | `readline` | `write` | `writeline`

Use Callbacks for TCP/IP Communication

You can enhance the power and flexibility of your TCP/IP client by using events and callbacks. An event occurs after a condition is met and can result in one or more callbacks.

While MATLAB is connected to a remote host with a TCP/IP client, you can use events to display a message, display data, analyze data, and so on. You can control callbacks through callback properties and callback functions. All event types have an associated callback property. Callback functions are MATLAB functions that you write to suit your specific application needs. Execute a callback when a particular event occurs by specifying the name of the callback function as the value for the associated callback property.

For an example of configuring callbacks with the `tcpclient` object, see “Communicate Binary and ASCII Data to an Echo Server Using TCP/IP” on page 22-124.

The `tcpclient` properties and functions associated with callbacks follow.

Property or Function	Purpose
`NumBytesAvailable`	Number of bytes available to read
`BytesAvailableFcn`	Bytes available callback function
`BytesAvailableFcnCount`	Number of bytes of data to trigger callback
`BytesAvailableFcnMode`	Bytes available callback trigger mode
`configureCallback`	Set callback function and trigger condition for communication with remote host over TCP/IP
`ErrorOccurredFcn`	Callback function triggered by error event
`UserData`	General purpose property for user data

See Also

More About

- “Communicate Binary and ASCII Data to an Echo Server Using TCP/IP” on page 22-124

Configure Connection in TCP/IP Explorer

After you select **Device > TCP/IP Connection** from the **TCP/IP Explorer** toolstrip, the **Configure** tab opens. Set the following connection parameters for the TCP/IP server that you want to connect to.

- **Address** — Server name or IP address.
- **Port** — Server port, specified as a number between 1 and 65535, inclusive.
- **Connect Timeout** — Allowed time in seconds to connect to the server, specified as a numeric value. This parameter specifies the maximum time to wait for a connection request to the specified server to succeed or fail.

After you have specified values for these parameters, click **Confirm Parameters**. If the connection is successful, your TCP/IP Connection appears in the **Device List**.

Go back to the **Devices** tab at any time by clicking **Cancel**.

See Also

Events and Callbacks

In this section...

"Event Types and Callback Properties" on page 7-12
"Responding To Event Information" on page 7-12
"Using Events and Callbacks" on page 7-14

Event Types and Callback Properties

The event types and associated callback properties supported by TCP/IP and UDP objects are listed below.

TCP/IP and UDP Event Types and Callback Properties

Event Type	Associated Properties
Bytes available	BytesAvailableFcn
	BytesAvailableFcnCount
	BytesAvailableFcnMode
Datagram received	DatagramReceivedFcn (UDP objects only)
Error	ErrorFcn
Output empty	OutputEmptyFcn
Timer	TimerFcn
	TimerPeriod

The datagram-received event is described below. For a description of the other event types, refer to "Event Types and Callback Properties" on page 5-24.

Note You cannot use ASCII values larger than 127 characters with `fgetl`, `fgets`, or `BytesAvailableFnc`. The functions are limited to 127 binary characters.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Datagram-Received Event

A datagram-received event is generated immediately after a complete datagram is received in the input buffer.

This event executes the callback function specified for the `DatagramReceivedFcn` property. It can be generated for both synchronous and asynchronous read operations.

Responding To Event Information

You can respond to event information in a callback function or in a record file. Event information stored in a callback function uses two fields: `Type` and `Data`. The `Type` field contains the event type,

while the Data field contains event-specific information. These two fields are associated with a structure that you define in the callback function header. Refer to “Debugging: Recording Information to Disk” on page 17-5 to learn about storing event information in a record file.

The event types and the values for the Type and Data fields are given below.

TCP/IP and UDP Event Information

Event Type	Field	Field Value
Bytes available	Type	BytesAvailable
	Data.AbsTime	day-month-year hour:minute:second
Datagram received	Type	DatagramReceived
	Data.AbsTime	day-month-year hour:minute:second
	Data.DatagramAddress	IP address character vector
	Data.DatagramLength	Number of bytes received as double
	Data.DatagramPort	Port number of sender as double
Error	Type	Error
	Data.AbsTime	day-month-year hour:minute:second
	Data.Message	An error string
Output empty	Type	OutputEmpty
	Data.AbsTime	day-month-year hour:minute:second
Timer	Type	Timer
	Data.AbsTime	day-month-year hour:minute:second

The Data field values are described below.

AbsTime Field

AbsTime is defined for all events, and indicates the absolute time the event occurred. The absolute time is returned using the MATLAB Command window clock format.

day-month-year hour:minute:second (7-1)

DatagramAddress Field

DatagramAddress is the IP address of the datagram sender.

DatagramLength Field

DatagramLength is the length of the datagram in bytes.

DatagramPort Field

DatagramPort is the sender's port number from which the datagram originated.

Message Field

Message is used by the error event to store the descriptive message that is generated when an error occurs.

Using Events and Callbacks

This example extends “UDP Communication Between Two Hosts” on page 8-5 to include a datagram received callback. The callback function is `instrcallback`, which displays information to the command line indicating that a datagram has been received.

The following command configures the callback for the UDP object `u2`.

```
u2.DatagramReceivedFcn = @instrcallback;
```

When a datagram is received, the following message is displayed.

```
DatagramReceived event occurred at 10:26:20 for the object:
UDP-doetom.dhpc.
25 bytes were received from address 192.168.1.12, port 8844.
```

Note You cannot use ASCII values larger than 127 characters with `fgetl`, `fgets`, or `BytesAvailableFnc`. The functions are limited to 127 binary characters.

Rules for Completing Read and Write Operations over TCP/IP and UDP

The rules for completing synchronous and asynchronous read and write operations are described here.

For a general overview about writing and reading data, as well as a list of all associated functions and properties, refer to "Communicating with Your Instrument" on page 2-8.

Completing Write Operations

A write operation using `fprintf` or `fwrite` completes when one of these conditions is satisfied:

- The specified data is written.
- The time specified by the `Timeout` property passes.

In addition to these rules, you can stop an asynchronous write operation at any time with the `stopasync` function.

A text command is processed by the instrument only when it receives the required terminator. For TCP/IP and UDP objects, each occurrence of `\n` in the ASCII command is replaced with the `Terminator` property value. Because the default format for `fprintf` is `%s\n`, all commands written to the instrument will end with the `Terminator` value. The default value of `Terminator` is the line feed character. The terminator required by your instrument will be described in its documentation.

Completing Read Operations

A read operation with `fgetl`, `fgets`, `fscanf`, or `readasync` completes when one of these conditions is satisfied:

- The terminator specified by the `Terminator` property is read. For UDP objects, `DatagramTerminateMode` must be `off`.
- The time specified by the `Timeout` property passes.
- The input buffer is filled.
- The specified number of values is read (`fscanf` and `readasync` only). For UDP objects, `DatagramTerminateMode` must be `off`.
- A datagram is received (for UDP objects, only when `DatagramTerminateMode` is `on`).

A read operation with `fread` completes when one of these conditions is satisfied:

- The time specified by the `Timeout` property passes.
- The input buffer is filled.
- The specified number of values is read. For UDP objects, `DatagramTerminateMode` must be `off`.
- A datagram is received (for UDP objects, only when `DatagramTerminateMode` is `on`).

Note Set the terminator property to `''` (null), if appropriate, to ensure efficient throughput of binary data.

In addition to these rules, you can stop an asynchronous read operation at any time with the `stopasync` function.

Basic Workflow to Read and Write Data over TCP/IP

This example illustrates how to use text and binary read and write operations with a TCP/IP object connected to a remote instrument. In this example, you create a vector of waveform data in the MATLAB workspace, upload the data to the instrument, and then read back the waveform.

The instrument is a Sony/Tektronix AWG520 Arbitrary Waveform Generator (AWG). Its address is `sonytekawg.yourdomain.com` and its port is 4000. The AWG's host IP address is 192.168.1.10 and is user configurable in the instrument. The associated host name is given by your network administrator. The port number is fixed and is found in the instrument's documentation:

1. **Create an instrument object** — Create a TCP/IP object associated with the AWG.

   ```
   t = tcpip('sonytekawg.yourdomain.com',4000);
   ```
2. **Connect to the instrument** — Before establishing a connection, the `OutputBufferSize` must be large enough to hold the data being written. In this example, 2577 bytes are written to the instrument. Therefore, the `OutputBufferSize` is set to `3000`.

   ```
   t.OutputBufferSize = 3000
   ```

   You can now connect `t` to the instrument.

   ```
   fopen(t)
   ```
3. **Write and read data** — Since the instrument's byte order is little-endian, configure the `ByteOrder` property to `littleEndian`.

   ```
   t.ByteOrder = 'littleEndian'
   ```

   Create the sine wave data.

   ```
   x = (0:499).*8*pi/500;
   data = sin(x);
   marker = zeros(length(data),1);
   marker(1) = 3;
   ```

   Instruct the instrument to write the file `sin.wfm` with Waveform File format, a total length of 2544 bytes, and a combined data and marker length of 2500 bytes.

   ```
   fprintf(t,'%s',['MMEMORY:DATA "sin.wfm",#42544MAGIC 1000' 13 10])
   fprintf(t,'%s','#42500')
   ```

   Write the sine wave to the instrument.

   ```
   for i = 1:length(data)
       fwrite(t,data(i),'float32');
       fwrite(t,marker(i));
   end
   ```

   Instruct the instrument to use a clock frequency of 100 MS/s for the waveform.

   ```
   fprintf(t,'%s',['CLOCK 1.0000000000e+008' 13 10 10])
   ```

   Read the waveform stored in the function generator's hard drive. The waveform contains 2000 bytes plus markers, header, and clock information. To store this data, close the connection and configure the input buffer to hold 3000 bytes.

   ```
   fclose(t)
   t.InputBufferSize = 3000
   ```

Reopen the connection to the instrument.

```
fopen(t)
```

Read the file `sin.wfm` from the function generator.

```
fprintf(t,'MMEMORY:DATA? "sin.wfm" ')
data = fread(t,t.BytesAvailable);
```

The next set of commands reads the same waveform as a `float32` array. To begin, write the waveform to the AWG.

```
fprintf(t,'MMEMORY:DATA? "sin.wfm" ')
```

Read the file header as ASCII characters.

```
header1 = fscanf(t)
header1 =
#42544MAGIC 1000
```

Read the next six bytes, which specify the length of data.

```
header2 = fscanf(t,'%s',6)
header2 =
#42500
```

Read the waveform using `float32` precision and read the markers using `uint8` precision. Note that one `float32` value consists of four bytes. Therefore, the following commands read 2500 bytes.

```
data = zeros(500,1);
marker = zeros(500,1);
for i = 1:500
    data(i) = fread(t,1,'float32');
    marker(i) = fread(t,1,'uint8');
end
```

Read the remaining data, which consists of clock information and termination characters.

```
clock = fscanf(t);
cleanup = fread(t,2);
```

4 **Disconnect and clean up** — When you no longer need `t`, you should disconnect it from the host, and remove it from memory and from the MATLAB workspace.

```
fclose(t)
delete(t)
clear t
```

Read and Write ASCII Data over TCP/IP

In this section...

"Functions and Properties" on page 7-19
"Configuring and Connecting to the Server" on page 7-20
"Writing ASCII Data" on page 7-20
"ASCII Write Properties" on page 7-21
"Reading ASCII Data" on page 7-21
"ASCII Read Properties" on page 7-22
"Cleanup" on page 7-22

This section provides details and examples exploring ASCII read and write operations with a TCP/IP object.

Note Most bench-top instruments (oscilloscopes, function generators, etc.) that provide network connectivity do not use raw TCP socket communication for instrument command and control. Instead, it is supported through the VISA standard. For more information on using VISA to communicate with your instrument, see "Get Started with VISA" on page 5-2.

Functions and Properties

These functions are used when reading and writing text:

Function	Purpose
`fprintf`	Write text to the server.
`fscanf`	Read data from the server and format as text.

These properties are associated with reading and writing text:

Property	Purpose
`ValuesReceived`	Specifies the total number of values read from the server.
`ValuesSent`	Specifies the total number of values sent to the server.
`InputBufferSize`	Specifies the total number of bytes that can be queued in the input buffer at one time.
`OutputBufferSize`	Specifies the total number of bytes that can be queued in the output buffer at one time.
`Terminator`	Character used to terminate commands sent to the server.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the TCP/IP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echotcpip('on', 4000)
```

You need to create a TCP/IP object. In this example, create a TCP/IP object associated with the host 127.0.0.1 (your local machine), port 4000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
t = tcpip('127.0.0.1', 4000);
```

Before you can perform a read or write operation, you must connect the TCP/IP object to the server with the `fopen` function.

```
fopen(t)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
t.Status
ans =
    open
```

Writing ASCII Data

You use the `fprintf` function to write ASCII data to the server.

```
fprintf(t, 'Hello World 123');
```

By default, the `fprintf` function operates in a synchronous mode. This means that `fprintf` blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the `Timeout` property

By default the `fprintf` function writes ASCII data using the `%s\n` format. All occurrences of `\n` in the command being written to the server are replaced with the `Terminator` property value. When using the default format, `%s\n`, all commands written to the server will end with the `Terminator` character.

For the previous command, the linefeed (LF) is sent after `'Hello World 123'` is written to the server, thereby indicating the end of the command.

You can also specify the format of the command written by providing a third input argument to `fprintf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s.

For example, the data command previously shown can be written to the server using three calls to `fprintf`.

```
fprintf(t, '%s', 'Hello');
fprintf(t, '%s', ' World');
fprintf(t, '%s\n', ' 123');
```

The `Terminator` character indicates the end of the command and is sent after the last call to `fprintf`.

ASCII Write Properties

The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the server at once. By default, `OutputBufferSize` is `512`.

```
t.OutputBufferSize
ans =
    512
```

The `ValuesSent` property indicates the total number of values written to the server since the object was connected to the server.

```
t.ValuesSent
ans =
    32
```

Reading ASCII Data

You use the `fscanf` function to read ASCII data from the server. For example, to read back the data returned from the echo server for our first `fprintf` command:

```
data = fscanf(t)
data =
    Hello World 123
```

By default, the `fscanf` function reads data using the `'%c'` format and blocks the MATLAB command line until one of the following occurs:

- The terminator is received as specified by the `Terminator` property
- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled
- The specified number of values is read

You can also specify the format of the data read by providing a second input argument to `fscanf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s.

The following commands return a numeric value as a double.

Clear anything still in the input buffer from the previous commands.

```
flushinput(t);
```

Send the data to the server.

```
fprintf(t, '0.8000');
```

Read the response.

```
data = fscanf(t, '%f')
data =
    0.8000
```

```
isnumeric(data)
ans =
    1
```

ASCII Read Properties

The `InputBufferSize` property specifies the maximum number of bytes you can read from the server. By default, `InputBufferSize` is `512`.

```
t.InputBufferSize
ans =
    512
```

The `ValuesReceived` property indicates the total number of values read from the server, including the terminator.

```
t.ValuesReceived
ans =
    32
```

Cleanup

If you are finished with the TCP/IP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(t);
delete(t);
clear t

echotcpip('off');
```

Read and Write Binary Data over TCP/IP

In this section...

"Functions and Properties" on page 7-23
"Configuring and Connecting to the Server" on page 7-24
"Writing Binary Data" on page 7-25
"Binary Write Properties" on page 7-25
"Configuring InputBufferSize" on page 7-25
"Reading Binary Data" on page 7-26
"Cleanup" on page 7-26

This section provides details and examples exploring binary read and write operations with a TCP/IP object.

Note Most bench-top instruments (oscilloscopes, function generators, etc.) that provide network connectivity do not use raw TCP socket communication for instrument command and control. Instead, it is supported through the VISA standard. For more information on using VISA to communicate with your instrument, see "Get Started with VISA" on page 5-2.

Functions and Properties

These functions are used when reading and writing binary data:

Function	Purpose
`fread`	Read binary data from the server.
`fwrite`	Write binary data to the server.

These properties are associated with reading and writing binary data:

Property	Purpose
`ValuesReceived`	Specifies the total number of values read from the server.
`ValuesSent`	Specifies the total number of values sent to the server.
`InputBufferSize`	Specifies the total number of bytes that can be queued in the input buffer at one time.
`OutputBufferSize`	Specifies the total number of bytes that can be queued in the output buffer at one time.
`ByteOrder`	Specifies the byte order of the server.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the TCP/IP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echotcpip('on', 4000)
```

You need to create a TCP/IP object. In this example, create a TCP/IP object associated with the host 127.0.0.1 (your local machine), port 4000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
t = tcpip('127.0.0.1', 4000);
```

You may need to configure the `OutputBufferSize` of the TCP/IP object. The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the server at once. By default, `OutputBufferSize` is `512`.

```
t.OutputBufferSize
ans =
    512
```

If the command specified in `fwrite` contains more than 512 bytes, an error is returned and no data is written to the server. In this example 4000 bytes will be written to the server. Therefore, the `OutputBufferSize` is increased to `4000`.

```
t.OutputBufferSize = 4000;
t.OutputBufferSize
ans =
    4000
```

You may need to configure the `ByteOrder` of the TCP/IP object. The `ByteOrder` property specifies the byte order of the server. By default `ByteOrder` is `bigEndian`.

```
t.ByteOrder
ans =
    'bigEndian'
```

If the server's byte order is little-endian, the `ByteOrder` property of the object can be configured to `littleEndian`:

```
t.ByteOrder = 'littleEndian'
t.ByteOrder
ans =
    'littleEndian'
```

Before you can perform a read or write operation, you must connect the TCP/IP object to the server with the `fopen` function.

```
fopen(t)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
t.Status
ans =
    open
```

Writing Binary Data

You use the `fwrite` function to write binary data to the server. For example, the following command will send a sine wave to the server.

Construct the sine wave to be written to the server.

```
x = (0:999) .* 8 * pi / 1000;
data = sin(x);
```

Write the sine wave to the server.

```
fwrite(t, data, 'float32');
```

By default, the `fwrite` function operates in a synchronous mode. This means that `fwrite` blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the `Timeout` property

By default the `fwrite` function writes binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fwrite`.

Note When performing a write operation, you should think of the transmitted data in terms of values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

Binary Write Properties

The `ValuesSent` property indicates the total number of values written to the server since the object was connected to the server.

```
t.ValuesSent
ans =
    1000
```

Configuring InputBufferSize

The `InputBufferSize` property specifies the maximum number of bytes that you can read from the server. By default, `InputBufferSize` is `512`.

```
t.InputBufferSize
ans =
    512
```

Next, the waveform stored in the function generator's memory will be read. The waveform contains 4000 bytes. Configure the `InputBufferSize` to hold 4000 bytes. Note, the `InputBufferSize` can be configured only when the object is not connected to the server.

```
fclose(t);
t.InputBufferSize = 4000;
t.InputBufferSize
```

```
ans =
    4000
```

Now that the property is configured correctly, you can reopen the connection to the server:

```
fopen(t);
```

Reading Binary Data

You use the `fread` function to read binary data from the server.

The `fread` function blocks the MATLAB command line until one of the following occurs:

- A timeout occurs as specified by the `Timeout` property
- The specified number of values is read
- The `InputBufferSize` number of values is read

By default the `fread` function reads binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fread`.

Note When performing a read operation, you should think of the received data in terms of values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

For reading `float32` binary data, send the waveform again. Closing the object clears any available data from earlier writes.

```
fwrite(t, data, 'float32');
```

Now read the same waveform as a `float32` array.

```
data = fread(t, 1000, 'float32');
```

The `ValuesReceived` property indicates the total number of values read from the server.

```
t.ValuesReceived
ans =
    1000
```

Cleanup

If you are finished with the TCP/IP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(t);
delete(t);
clear t

echotcpip('off');
```

Asynchronous Read and Write Operations over TCP/IP

In this section...

"Functions and Properties" on page 7-27
"Synchronous Versus Asynchronous Operations" on page 7-28
"Configuring and Connecting to the Server" on page 7-28
"Reading Data Asynchronously" on page 7-28
"Reading Data Asynchronously – Continuous ReadAsyncMode" on page 7-29
"Reading Data Asynchronously – Manual ReadAsyncMode" on page 7-29
"Defining an Asynchronous Read Callback" on page 7-30
"Using Callbacks During an Asynchronous Read" on page 7-30
"Writing Data Asynchronously" on page 7-31
"Cleanup" on page 7-31

This section provides details and examples exploring asynchronous read and write operations with a TCP/IP object.

Note Most bench-top instruments (oscilloscopes, function generators, etc.) that provide network connectivity do not use raw TCP socket communication for instrument command and control. Instead, it is supported through the VISA standard. For more information on using VISA to communicate with your instrument, see "Get Started with VISA" on page 5-2.

Functions and Properties

These functions are associated with reading and writing text asynchronously:

Function	Purpose
`fprintf`	Write text to a server.
`readasync`	Asynchronously read bytes from a server.
`stopasync`	Stop an asynchronous read or write operation.

These properties are associated with reading and writing text asynchronously:

Property	Purpose
`BytesAvailable`	Indicates the number of bytes available in the input buffer.
`TransferStatus`	Indicates what type of asynchronous operation is in progress.
`ReadAsyncMode`	Indicates whether data is read continuously in the background or whether you must call the `readasync` function to read data asynchronously.

Additionally, you can use all the callback properties during asynchronous read and write operations.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For

information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Synchronous Versus Asynchronous Operations

The object can operate in synchronous mode or in asynchronous mode. When the object is operating synchronously, the read and write routines block the MATLAB command line until the operation has completed or a timeout occurs. When the object is operating asynchronously, the read and write routines return control immediately to the MATLAB command line.

Additionally, you can use callback properties and callback functions to perform tasks as data is being written or read. For example, you can create a callback function that notifies you when the read or write operation has finished.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the TCP/IP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echotcpip('on', 4000)
```

You need to create a TCP/IP object. In this example, create a TCP/IP object associated with the host 127.0.0.1 (your local machine), port 4000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
t = tcpip('127.0.0.1', 4000);
```

Before you can perform a read or write operation, you must connect the TCP/IP object to the server with the `fopen` function.

```
fopen(t)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
t.Status
ans =
    open
```

Reading Data Asynchronously

You can read data asynchronously with the TCP/IP object in one of these two ways:

- Continuously, by setting `ReadAsyncMode` to `continuous`. In this mode, data is automatically stored in the input buffer as it becomes available from the server.
- Manually, by setting `ReadAsyncMode` to `manual`. In this mode, you must call the `readasync` function to store data in the input buffer.

The `fscanf`, `fread`, `fgetl` and `fgets` functions are used to bring the data from the input buffer into MATLAB. These functions operate synchronously.

Reading Data Asynchronously – Continuous ReadAsyncMode

To begin, read data continuously.

```
t.ReadAsyncMode = continuous;
```

Now, send data to the server that will be returned for reading.

```
fprintf(t, 'Hello World 123');
```

Because the `ReadAsyncMode` property is set to `continuous`, the object is continuously checking whether any data is available. Once the last `fprintf` function completes, the server begins sending data, the data is read from the server and is stored in the input buffer.

```
t.BytesAvailable
ans =
    16
```

You can bring the data from the object's input buffer into the MATLAB workspace with `fscanf`.

```
fscanf(t)
ans =
    Hello World 123
```

Reading Data Asynchronously – Manual ReadAsyncMode

Next, read data manually.

```
t.ReadAsyncMode = manual;
```

Now, send data to the server that will be returned for reading.

```
fprintf(t, 'Hello World 456');
```

Once the last `fprintf` function completes, the server begins sending data. However, because `ReadAsyncMode` is set to `manual`, the object is not reading the data being sent from the server. Therefore no data is being read and placed in the input buffer.

```
t.BytesAvailable
ans =
    0
```

The `readasync` function can asynchronously read the data from the server. The `readasync` function returns control to the MATLAB command line immediately.

The `readasync` function takes two input arguments. The first argument is the server object and the second argument is the `size`, the amount of data to be read from the server.

The `readasync` function without a `size` specified assumes `size` is given by the difference between the `InputBufferSize` property value and the `BytesAvailable` property value. The asynchronous read terminates when:

- The terminator is read as specified by the `Terminator` property
- The specified number of bytes have been read
- A timeout occurs as specified by the `Timeout` property

- The input buffer is filled

An error event will be generated if `readasync` terminates due to a timeout.

The object starts querying the server for data when the `readasync` function is called. Because all the data was sent before the `readasync` function call, no data will be stored in the input buffer and the data is lost.

When the TCP/IP object is in manual mode (the `ReadAsyncMode` property is configured to `manual`), data that is sent from the server to the computer is not automatically stored in the input buffer of the TCP/IP object. Data is not stored until `readasync` or one of the blocking read functions is called.

Manual mode should be used when a stream of data is being sent from your server and you only want to capture portions of the data.

Defining an Asynchronous Read Callback

You can configure a TCP/IP object to notify you when a terminator has been read using the `dispcallback` function.

```
t.ReadAsyncMode = 'continuous';
t.BytesAvailableFcn = 'dispcallback';
```

Note, the default value for the `BytesAvailableFcnMode` property indicates that the callback function defined by the `BytesAvailableFcn` property will be executed when the terminator has been read.

The callback function `dispcallback` displays event information for the specified event. Using the syntax `dispcallback(obj, event)`, it displays a message containing the type of event, the name of the object that caused the event to occur, and the time the event occurred.

```
callbackTime = datestr(datenum(event.Data.AbsTime));
fprintf(['A ' event.Type ' event occurred for ' obj.Name ' at '
     callbackTime '.\n']);
```

Using Callbacks During an Asynchronous Read

Once the terminator is read from the server and placed in the input buffer, `dispcallback` is executed and a message is posted to the MATLAB command window indicating that a `BytesAvailable` event occurred.

```
fprintf(t, 'Hello World 789')
t.BytesAvailable
ans =
    16

data = fscanf(t, '%c', 18)
data =
    Hello World 789
```

Note If you need to stop an asynchronous read or write operation, you do not have to wait for the operation to complete. You can use the `stopasync` function to stop the asynchronous read or write.

Writing Data Asynchronously

You can perform an asynchronous write with the `fprintf` or `fwrite` functions by passing `'async'` as the last input argument.

In asynchronous mode, you can use callback properties and callback functions to perform tasks while data is being written. For example, configure the object to notify you when an asynchronous write operation completes.

```
t.OutputEmptyFcn = 'dispcallback';
fprintf(t, 'Hello World 123', 'async')
```

Note If you need to stop an asynchronous read or write operation, you do not have to wait for the operation to complete. You can use the `stopasync` function to stop the asynchronous read or write.

Cleanup

If you are finished with the TCP/IP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(t);
delete(t);
clear t
```

```
echotcpip('off');
```

TCP/IP and UDP Comparison

Transmission Control Protocol (TCP or TCP/IP) and User Datagram Protocol (UDP or UDP/IP) are both transport protocols layered on top of the Internet Protocol (IP). Use the TCP/IP and UDP interfaces for reading and writing both binary data and ASCII data. You can read and write to servers, computers, instruments, and use applications such as streaming video and audio, point of sale systems, and other business applications.

Supported Platforms

The TCP/IP and UDP interfaces are supported on the following platforms.

- Linux
- macOS
- Windows 10

Interface Comparison

TCP/IP and UDP are compared below:

- **Connection Versus Connectionless** — TCP/IP is a connection-based protocol, while UDP is a connectionless protocol. In TCP/IP, the two ends of the communication link must be connected at all times during the communication. An application using UDP prepares a packet and sends it to the receiver's address without first checking to see if the receiver is ready to receive a packet. If the receiving end is not ready to receive a packet, the packet is lost.
- **Stream Versus Packet** — TCP/IP is a stream-oriented protocol, while UDP is a packet-oriented protocol. This means that TCP/IP is considered to be a long stream of data that is transmitted from one end of the connection to the other end, and another long stream of data flowing in the opposite direction. The TCP/IP stack is responsible for breaking the stream of data into packets and sending those packets while the stack at the other end is responsible for reassembling the packets into a data stream using information in the packet headers. UDP, on the other hand, is a packet-oriented protocol where the application itself divides the data into packets and sends them to the other end. The other end does not have to reassemble the data into a stream. Note, some applications might present the data as a stream when the underlying protocol is UDP. However, this is the layering of an additional protocol on top of UDP, and it is not something inherent in the UDP protocol itself.
- **TCP/IP Is a Reliable Protocol, While UDP Is Unreliable** — The packets that are sent by TCP/IP contain a unique sequence number. The starting sequence number is communicated to the other side at the beginning of communication. The receiver acknowledges each packet, and the acknowledgment contains the sequence number so that the sender knows which packet was acknowledged. This implies that any packets lost on the way can be retransmitted (the sender would know that they did not reach their destination because it had not received an acknowledgment). Also, packets that arrive out of sequence can be reassembled in the proper order by the receiver.

  Further, timeouts can be established because the sender knows (from the first few packets) how long it takes on average for a packet to be sent and its acknowledgment received. UDP, on the other hand, sends the packets and does not keep track of them. Thus, if packets arrive out of sequence, or are lost in transmission, the receiving end (or the sending end) has no way of knowing.

Note that "unreliable" is used in the sense of "not guaranteed to succeed" as opposed to "will fail a lot of the time." In practice, UDP is quite reliable as long as the receiving socket is active and is processing data as quickly as it arrives.

See Also

Related Examples

- “Create TCP/IP Client and Configure Settings” on page 7-3
- “Create a UDP Object and View Properties” on page 8-2

Communicate Using TCP/IP Server Sockets

In this section...
“About Server Sockets” on page 7-34
“Communicate Between Two Instances of MATLAB” on page 7-34

About Server Sockets

Support for server sockets is available using the `tcpserver` function. This support is for a single remote connection. You can use this connection to communicate between a client and MATLAB or between two instances of MATLAB.

For example, you might collect data such as a waveform into one instance of MATLAB and then transfer it to another instance of MATLAB.

Note The use of the server socket on either the client or server side should be done in accordance with the license agreement as it relates to your particular license option and activation type. If you have questions, you should consult with the administrator for your license or your legal department.

This is intended for use behind a firewall on a private network.

Communicate Between Two Instances of MATLAB

The following example shows how to connect two MATLAB sessions on the same computer, showing the example code for each session. To use two different computers, replace `"localhost"` with the IP address of the server in the code for Session 2. Using 0.0.0.0 as the IP address means that the server will accept the first machine that tries to connect. To restrict the connections that will be accepted, replace `"0.0.0.0"` with the address of the client in the code for Session 1.

Session 1: MATLAB Server

Accept a connection from any machine on port 30000.

```
server = tcpserver("0.0.0.0",30000)
```

```
server = 

  TCPServer with properties:

        ServerAddress: "0.0.0.0"
           ServerPort: 30000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Session 2: MATLAB Client

This code is running on a second instance of MATLAB.

Create a client interface that connects to the server.

```
client = tcpclient("localhost",30000)
```

```
client = 

  tcpclient with properties:

              Address: 'localhost'
                 Port: 30000
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a waveform and visualize it.

```
data = sin(1:64);
plot(data);
```

Write the waveform to the client. Since the client is connected to the server, this data is available in the server session.

```
write(client,data,"double")
```

Session 1: MATLAB Server

Read the waveform and confirm it visually by plotting it.

```
data = read(server,server.NumBytesAvailable,"double");
plot(data);
```

See Also

tcpclient | tcpserver

Related Examples

- “Communicate Between a TCP/IP Client and Server in MATLAB” on page 22-127
- “Read Data from Arduino Using TCP/IP Communication” on page 22-132

Transition Your Code to tcpclient Interface

The `tcpip` function, its object functions, and its properties will be removed. Use the `tcpclient` interface instead.

`tcpip` Interface	`tcpclient` Interface	Example
`tcpip`	`tcpclient`	"Create a TCP/IP Client" on page 7-36
`fwrite` and `fread`	`write` and `read`	"Write and Read" on page 7-37
`fprintf`	`writeline`	"Read Terminated String" on page 7-37
`fscanf`, `fgetl`, and `fgets`	`readline`	"Read Terminated String" on page 7-37
		"Read and Parse String Data" on page 7-38
`query`	`writeread`	"Write and Read Back Data" on page 7-39
`binblockwrite` and `binblockread`	`writebinblock` and `readbinblock`	"Write and Read Data with the Binary Block Protocol" on page 7-39
`flushinput` and `flushoutput`	`flush`	"Flush Data from Memory" on page 7-40
`Terminator`	`configureTerminator`	"Set Terminator" on page 7-40
`BytesAvailableFcnCount`, `BytesAvailableFcnMode`, `BytesAvailableFcn`, and `BytesAvailable`	`configureCallback`	"Set Up Callback Function" on page 7-40
`tcpip` Properties	`tcpclient` Properties on page 24-348	

Create a TCP/IP Client

These examples show how to create and clear a TCP/IP client using the recommended functionality.

Functionality	Use This Instead
`t = tcpip("localhost",3030);` `fopen(t)`	`t = tcpclient("localhost",3030);` `t.ByteOrder = "big-endian";`
`t = tcpip("127.0.0.1",3030,"NetworkRole","client");` `fopen(t)`	`t = tcpclient("127.0.0.1",3030);` `t.ByteOrder = "big-endian";`

`fclose(t)`
`delete(t)`
`clear t`
	`clear t`

Note Since the default value of `ByteOrder` is `bigEndian` for `tcpip` objects and `little-endian` for `tcpclient` objects, you must set it using dot notation to make the property values match.

For more information, see `tcpclient`.

Write and Read

These examples use an echo server to show how to perform a binary write and read, and how to write and read nonterminated string data, using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>echotcpip("on",3030)

% t is a tcpip object
fwrite(t,1:5);
data = fread(t,5)

data =

     1
     2
     3
     4
     5</pre></td><td><pre>echotcpip("on",3030)

% t is a tcpclient object
write(t,1:5,"uint8")
data = read(t,5)

data =

  1×5 uint8 row vector

   1   2   3   4   5

data = double(data)

data =

     1   2   3   4   5</pre></td></tr>
<tr><td><pre>echotcpip("on",3030)

% t is a tcpip object
fwrite(t,"hello","char")
length = 5;
data = fread(t,length,"char")

data =

   104
   101
   108
   108
   111

data = char(data)'

data =

    'hello'</pre></td><td><pre>echotcpip("on",3030)

% t is a tcpclient object
write(t,"hello","string");
length = 5;
data = read(t,length,"string")

data =

    "hello"</pre></td></tr>
</table>

For more information, see `write` or `read` .

Read Terminated String

These examples show how to write and read terminated string data using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>echotcpip("on",3030)

% t is a tcpip object
t.Terminator = "CR";
fprintf(t,"hello")
data = fscanf(t)

data =

    'hello
     '</pre></td><td rowspan="3"><pre>echotcpip("on",3030)

% t is a tcpclient object
configureTerminator(t,"CR");
writeline(t,"hello");
data = readline(t)

a =

    "hello"</pre></td></tr>
<tr><td><pre>echotcpip("on",3030)

% t is a tcpip object
t.Terminator = "CR";
fprintf(t,"hello")
data = fgetl(t)

data =

    'hello'</pre>
<code>fgetl</code> reads until the specified terminator is reached and then discards the terminator.</td></tr>
<tr><td><pre>echotcpip("on",3030)

% t is a tcpip object
t.Terminator = "CR";
fprintf(t,"hello")
data = fgets(t)

data =

    'hello
     '</pre>
<code>fgets</code> reads until the specified terminator is reached and then returns the terminator.</td></tr>
</table>

For more information, see `writeline` or `readline`.

Read and Parse String Data

This example shows how to read and parse string data using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>% t is a tcpip object
data = scanstr(t,';')

data =

  3×1 cell array

    {'a'}
    {'b'}
    {'c'}</pre></td><td><pre>% t is a tcpclient object
data = readline(t)

data =

    "a;b;c"

data = strsplit(data,";")

data =

  1×3 string array

    "a"   "b"   "c"</pre></td></tr>
</table>

For more information, see `readline`.

Write and Read Back Data

This example shows how to write ASCII terminated data and read ASCII terminated data back using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>% t is a tcpip object
data = query(t,'ctrlcmd')

data =

    'success'</pre></td><td><pre>% t is a tcpclient object
data = writeread(t,"ctrlcmd")

data =

    "success"</pre></td></tr>
</table>

For more information, see `writeread`.

Write and Read Data with the Binary Block Protocol

This example shows how to write data with the IEEE standard binary block protocol using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>% t is a tcpip object
binblockwrite(t,1:5);
data = binblockread(t)

data =

     1
     2
     3
     4
     5</pre></td><td><pre>% t is a tcpclient object
writebinblock(t,1:5,"uint8");
data = readbinblock(t)

data =

     1   2   3   4   5</pre></td></tr>
</table>

For more information, see `writebinblock` or `readbinblock`.

Flush Data from Memory

These examples show how to flush data from the buffer using the recommended functionality.

Functionality	Use This Instead
% t is a tcpip object flushinput(t)	% t is a tcpclient object flush(t,"input")
% t is a tcpip object flushoutput(t)	% t is a tcpclient object flush(t,"output")
% t is a tcpip object flushinput(t) flushoutput(t)	% t is a tcpclient object flush(t)

For more information, see `flush`.

Set Terminator

These examples show how to set the terminator using the recommended functionality.

Functionality	Use This Instead
% t is a tcpip object t.Terminator = "CR/LF";	% t is a tcpclient object configureTerminator(t,"CR/LF")
% t is a tcpip object t.Terminator = {"CR/LF" [10]};	% t is a tcpclient object configureTerminator(t,"CR/LF",10)

For more information, see `configureTerminator`.

Set Up Callback Function

These examples show how to set up a callback function using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>
% t is a tcpip object
t.BytesAvailableFcnCount = 5
t.BytesAvailableFcnMode = "byte"
t.BytesAvailableFcn = @mycallback

function mycallback(src,evt)
   data = fread(src,src.BytesAvailableFcnCount);
   disp(evt)
   disp(evt.Data)
end

    Type: 'BytesAvailable'
    Data: [1×1 struct]

    AbsTime: [2019 12 21 16 35 9.7032]
</pre></td><td><pre>
% t is a tcpclient object
configureCallback(t,"byte",5,@mycallback);

function mycallback(src,evt)
   data = read(src,src.BytesAvailableFcnCount);
   disp(evt)
end

  ByteAvailableInfo with properties:

    BytesAvailableFcnCount: 5
                   AbsTime: 21-Dec-2019 12:23:01
</pre></td></tr>
</table>

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>
% t is a tcpip object
t.Terminator = "CR"
t.BytesAvailableFcnMode = "terminator"
t.BytesAvailableFcn = @mycallback

function mycallback(src,evt)
   data = fscanf(src);
   disp(evt)
   disp(evt.Data)
end

    Type: 'BytesAvailable'
    Data: [1×1 struct]

    AbsTime: [2019 12 21 16 35 9.7032]
</pre></td><td><pre>
% t is a tcpclient object
configureTerminator(t,"CR")
configureCallback(t,"terminator",@mycallback);

function mycallback(src,evt)
   data = readline(src);
   disp(evt)
end

  TerminatorAvailableInfo with properties:

                 AbsTime: 21-Dec-2019 12:23:01
</pre></td></tr>
</table>

For more information, see `configureCallback`.

See Also

`tcpclient`

More About

- R2020a TCP/IP and UDP Interface Topics

Transition Your Code to tcpserver Interface

The `tcpip` function, its object functions, and its properties will be removed. Use the `tcpserver` interface instead.

tcpip Interface	tcpserver Interface	Example
`tcpip`	`tcpserver`	"Create a TCP/IP Server" on page 7-42
`fwrite` and `fread`	`write` and `read`	"Write and Read" on page 7-43
`fprintf`	`writeline`	"Read Terminated String" on page 7-43
`fscanf`, `fgetl`, and `fgets`	`readline`	"Read Terminated String" on page 7-43 "Read and Parse String Data" on page 7-44
`binblockwrite` and `binblockread`	`writebinblock` and `readbinblock`	"Write and Read Data with the Binary Block Protocol" on page 7-45
`flushinput` and `flushoutput`	`flush`	"Flush Data from Memory" on page 7-45
`Terminator`	`configureTerminator`	"Set Terminator" on page 7-46
`BytesAvailableFcnCount`, `BytesAvailableFcnMode`, `BytesAvailableFcn`, and `BytesAvailable`	`configureCallback`	"Set Up Callback Function" on page 7-46
`tcpip` Properties	`tcpserver` Properties on page 24-360	

Create a TCP/IP Server

These examples show how to create and clear a TCP/IP server using the recommended functionality.

Functionality	Use This Instead

`t = tcpip("192.168.2.15",3030,"NetworkRole","server");`
`fopen(t)`
 	
This binds to host `"0.0.0.0"` (internally) and port 3030 and only accepts client connections coming from `"192.168.2.15"`. MATLAB is blocked until a client connection is established.
	`t = tcpserver("0.0.0.0",3030);`
	
	This binds to `"0.0.0.0"` and port 3030. `"0.0.0.0"` means that it accepts any incoming client connection requests on port 3030. MATLAB is not blocked.

`fclose(t)`
`delete(t)`
`clear t`
	`clear t`

For more information, see `tcpserver`.

Write and Read

These examples show how to perform a binary write and read, and how to write and read nonterminated string data, using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>% t is a tcpip object
fwrite(t,1:5);
data = fread(t,5)

data =

     1
     2
     3
     4
     5</pre></td><td><pre>% t is a tcpserver object
write(t,1:5,"uint8")
data = read(t,5)

data =

     1   2   3   4   5</pre></td></tr>
<tr><td><pre>% t is a tcpip object
fwrite(t,"hello","char")
length = 5;
data = fread(t,length,"char")

data =

   104
   101
   108
   108
   111

data = char(data)'

data =

    'hello'</pre></td><td><pre>% t is a tcpserver object
write(t,"hello","string");
length = 5;
data = read(t,length,"string")

data =

    "hello"</pre></td></tr>
</table>

For more information, see `write` or `read` .

Read Terminated String

These examples show how to write and read terminated string data using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td>% t is a tcpip object
t.Terminator = "CR";
fprintf(t,"hello")
data = fscanf(t)

data =

    'hello
     '</td><td rowspan="3">% t is a tcpserver object
configureTerminator(t,"CR");
writeline(t,"hello");
data = readline(t)

data =

    "hello"</td></tr>
<tr><td>% t is a tcpip object
t.Terminator = "CR";
fprintf(t,"hello")
data = fgetl(t)

data =

    'hello'

fgetl reads until the specified terminator is reached and then discards the terminator.</td></tr>
<tr><td>% t is a tcpip object
t.Terminator = "CR";
fprintf(t,"hello")
data = fgets(t)

data =

    'hello
     '

fgets reads until the specified terminator is reached and then returns the terminator.</td></tr>
</table>

For more information, see `writeline` or `readline`.

Read and Parse String Data

This example shows how to read and parse string data using the recommended functionality.

Functionality	Use This Instead

`% t is a tcpip object`
`data = scanstr(t,';')`
 	
`data =`
 	
`  3×1 cell array`
 	
`    {'a'}`
`    {'b'}`
`    {'c'}`
	`% t is a tcpserver object`
	`data = readline(t)`
	
	`data =`
	
	`    "a;b;c"`
	
	`data = strsplit(data,";")`
	
	`data =`
	
	`  1×3 string array`
	
	`    "a"    "b"    "c"`

For more information, see `readline`.

Write and Read Data with the Binary Block Protocol

This example shows how to write data with the IEEE standard binary block protocol using the recommended functionality.

Functionality	Use This Instead

`% t is a tcpip object`
`binblockwrite(t,1:5);`
`data = binblockread(t)`
 	
`data =`
 	
`     1`
`     2`
`     3`
`     4`
`     5`
	`% t is a tcpserver object`
	`writebinblock(t,1:5,"uint8");`
	`data = readbinblock(t)`
	
	`data =`
	
	`     1     2     3     4     5`

For more information, see `writebinblock` or `readbinblock`.

Flush Data from Memory

These examples show how to flush data from the buffer using the recommended functionality.

Functionality	Use This Instead
`% t is a tcpip object` `flushinput(t)`	`% t is a tcpserver object` `flush(t,"input")`
`% t is a tcpip object` `flushoutput(t)`	`% t is a tcpserver object` `flush(t,"output")`

`% t is a tcpip object`
`flushinput(t)`
`flushoutput(t)`
	`% t is a tcpserver object`
	`flush(t)`

For more information, see `flush`.

Set Terminator

These examples show how to set the terminator using the recommended functionality.

Functionality	Use This Instead
`% t is a tcpip object` `t.Terminator = "CR/LF";`	`% t is a tcpserver object` `configureTerminator(t,"CR/LF")`
`% t is a tcpip object` `t.Terminator = {"CR/LF" [10]};`	`% t is a tcpserver object` `configureTerminator(t,"CR/LF",10)`

For more information, see `configureTerminator`.

Set Up Callback Function

These examples show how to set up a callback function using the recommended functionality.

Functionality	Use This Instead

`% t is a tcpip object`
`t.BytesAvailableFcnCount = 5`
`t.BytesAvailableFcnMode = "byte"`
`t.BytesAvailableFcn = @mycallback`
 	
`function mycallback(src,evt)`
`   data = fread(src,src.BytesAvailableFcnCount);`
`   disp(evt)`
`   disp(evt.Data)`
`end`
 	
`    Type: 'BytesAvailable'`
`    Data: [1×1 struct]`
 	
`    AbsTime: [2019 12 21 16 35 9.7032]`
	`% t is a tcpserver object`
	`configureCallback(t,"byte",5,@mycallback);`
	
	`function mycallback(src,evt)`
	`   data = read(src,src.BytesAvailableFcnCount);`
	`   disp(evt)`
	`end`
	
	`  ByteAvailableInfo with properties:`
	
	`    BytesAvailableFcnCount: 5`
	`                   AbsTime: 21-Dec-2019 12:23:01`

`% t is a tcpip object`
`t.Terminator = "CR"`
`t.BytesAvailableFcnMode = "terminator"`
`t.BytesAvailableFcn = @mycallback`
 	
`function mycallback(src,evt)`
`   data = fscanf(src);`
`   disp(evt)`
`   disp(evt.Data)`
`end`
 	
`    Type: 'BytesAvailable'`
`    Data: [1×1 struct]`
 	
`    AbsTime: [2019 12 21 16 35 9.7032]`
	`% t is a tcpserver object`
	`configureTerminator(t,"CR")`
	`configureCallback(t,"terminator",@mycallback);`
	
	`function mycallback(src,evt)`
	`   data = readline(src);`
	`   disp(evt)`
	`end`
	
	`  TerminatorAvailableInfo with properties:`
	
	`                   AbsTime: 21-Dec-2019 12:23:01`

For more information, see `configureCallback`.

See Also

`tcpserver`

More About

- R2020a TCP/IP and UDP Interface Topics

Controlling Instruments Using UDP

- "Create a UDP Object and View Properties" on page 8-2
- "UDP Communication Between Two Hosts" on page 8-5
- "Write and Read ASCII Data Over UDP" on page 8-7
- "Write and Read Binary Data Over UDP" on page 8-9
- "Use Callbacks for UDP Communication" on page 8-13
- "Transition Your Code to udpport Interface" on page 8-18
- "Basic Workflow to Read and Write Data over UDP" on page 8-24
- "Asynchronous Read and Write Operations over UDP" on page 8-26

Create a UDP Object and View Properties

In this section...
"Create a Byte-Type UDP Object and View Properties" on page 8-2
"Create a Datagram-Type UDP Object and View Properties" on page 8-3
"Enable Port Sharing over UDP" on page 8-4

Create a UDP object with the `udpport` function. `udpport` does not require the name of the remote host as an input argument. Although UDP is a stateless connection, opening a UDP object with an invalid port number generates an error.

You can configure property values during object creation, such as the `LocalPort` property if you will use the object to read data from the instrument.

Create a Byte-Type UDP Object and View Properties

Create a UDP object associated with local port 3533. This creates a byte-type `udpport` object.

```
u = udpport("LocalPort",3533)
```

```
u = 

  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 3533
    NumBytesAvailable: 0

  Show all properties, functions
```

After you create a `udpport` object, you can view a full list of properties and their values. Click `properties` in the `udpport` output.

```
           IPAddressVersion: "IPV4"
                  LocalHost: "0.0.0.0"
                  LocalPort: 3533
          NumBytesAvailable: 0

                  ByteOrder: "little-endian"
                    Timeout: 10
                 Terminator: "LF"

          EnablePortSharing: 0
            EnableBroadcast: 0
            EnableMulticast: 0
    EnableMulticastLoopback: 1
             MulticastGroup: ""

      BytesAvailableFcnMode: "off"
     BytesAvailableFcnCount: 64
          BytesAvailableFcn: []
```

```
        OutputDatagramSize: 512
           NumBytesWritten: 0

          ErrorOccurredFcn: []
                  UserData: []
```

For more information about how to configure these properties, see “Properties” on page 24-377.

Create a Datagram-Type UDP Object and View Properties

You can also create a datagram-type `udpport` object by specifying the object type during object creation.

```
u = udpport("datagram")
```

```
u = 

  UDPPort with properties:

         IPAddressVersion: "IPV4"
                LocalHost: "0.0.0.0"
                LocalPort: 64655
    NumDatagramsAvailable: 0

  Show all properties, functions
```

Clicking the `properties` link for this object displays a different set of properties than the byte-type object.

```
  Show all properties, functions

              IPAddressVersion: "IPV4"
                     LocalHost: "0.0.0.0"
                     LocalPort: 64655
         NumDatagramsAvailable: 0

                     ByteOrder: "little-endian"
                       Timeout: 10

             EnablePortSharing: 0
               EnableBroadcast: 0
               EnableMulticast: 0
       EnableMulticastLoopback: 1
                MulticastGroup: ""

      DatagramsAvailableFcnMode: "off"
     DatagramsAvailableFcnCount: 1
         DatagramsAvailableFcn: []
            OutputDatagramSize: 512
           NumDatagramsWritten: 0

              ErrorOccurredFcn: []
                      UserData: []
```

Enable Port Sharing over UDP

UDP ports can be shared by other applications to allow for multiple applications to listen to the UDP datagrams on that port. You can bind a UDP object to a specific `LocalPort` number, and in another application bind a UDP socket to that same local port number so both can receive UDP broadcast or multicast data.

This allows for the ability to listen to UDP broadcasts or multicasts on the same local port number in both MATLAB and other applications. You can enable and disable this capability with `udpport` object property `EnablePortSharing`.

The `EnablePortSharing` property allows you to control UDP port sharing, and the possible values are `true` and `false`. The default value is `false`. This property can be set only during object creation using a name-value pair argument.

```
u = udpport("LocalPort",3030,"EnablePortSharing",true);
```

You can now do read and write operations, and other applications can access the port since port sharing is enabled.

See Also
`udpport`

More About
- “UDP Communication Between Two Hosts” on page 8-5
- “Write and Read ASCII Data Over UDP” on page 8-7
- “Write and Read Binary Data Over UDP” on page 8-9

UDP Communication Between Two Hosts

These are the minimum steps required to communicate between two hosts over UDP.

This example illustrates how you can use UDP objects to communicate between two dedicated hosts. In this example, you know the names of both hosts and the ports they use for communication with each other. One host has the name `doejohn.dhpc`, using local port `8844`, and the other host is `doetom.dhpc`, using local port `8866`.

1. **Create interface objects** — Create a UDP object on each host, referencing the other as the remote host.

   On host `doejohn.dhpc`, create `u1`. The object constructor specifies the name of the local port to use on the machine where this object is created.

   ```
   u1 = udpport("LocalPort",8844)
   ```

   ```
   u1 = 

     UDPPort with properties:

        IPAddressVersion: "IPV4"
               LocalHost: "0.0.0.0"
               LocalPort: 8844
       NumBytesAvailable: 0

     Show all properties, functions
   ```

   On host `doetom.dhpc`, create `u2`. The object constructor specifies the name of the local port and local host to use on the machine where this object is created.

   ```
   u2 = udpport("LocalPort",8866,"LocalHost","doetom.dhpc")
   ```

   ```
   u2 = 

     UDPPort with properties:

        IPAddressVersion: "IPV4"
               LocalHost: "172.31.42.41"
               LocalPort: 8866
       NumBytesAvailable: 0

     Show all properties, functions
   ```

2. **Write and read data** — Communication between the two hosts is now a matter of sending and receiving data. Write a message from `doejohn.dhpc` to `doetom.dhpc`.

   On host `doejohn.dhpc`, write data to the remote host via `u1`:

   ```
   write(u1,"Ready for data transfer.","string","doetom.dhpc",8866)
   ```

   On host `doetom.dhpc`, read data coming in from the remote host via `u2`:

   ```
   read(u2,u2.NumBytesAvailable,"string")
   ```

   ```
   ans = 
   ```

```
"Ready for data transfer."
```

3 **Disconnect and clean up** — When you no longer need `u1` on host `doejohn.dhpc`, you should clear the object.

```
clear u1
```

When you no longer need `u2`, clear the object on the host `doetom.dhpc`.

```
clear u2
```

See Also

`read` | `write`

More About

- “Write and Read ASCII Data Over UDP” on page 8-7
- “Write and Read Binary Data Over UDP” on page 8-9

Write and Read ASCII Data Over UDP

In this example, write and read ASCII data with a UDP object.

Configure and Connect to the Server

Use an echo server to experiment with the basic functionality of the UDP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echoudp("on",4040)
```

Create a byte-type `udpport` object. Datagram-type `udpport` objects do not support communication with ASCII-terminated data.

```
u = udpport
```

```
u = 

  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 53816
    NumBytesAvailable: 0

  Show all properties, functions
```

Write ASCII Data

Use the `writeline` function to write ASCII data to the server. Write a string to the echo server.

```
writeline(u,"Request Time","localhost",4040)
```

The function suspends MATLAB execution until all the data is written or a timeout occurs as specified by the `Timeout` property of the `udpport` object.

Check the default ASCII terminator.

```
u.Terminator
```

```
ans = 

    "LF"
```

The `writeline` function automatically appends the linefeed (LF) terminator to `"Request Time"` before it is written to the server, indicating the end of the command.

Read ASCII Data

Confirm the success of the write operation by viewing the `NumBytesAvailable` property.

```
u.NumBytesAvailable
```

```
ans =

    13
```

Since the `udpport` object is connected to an echo server, the data you write is returned to the object. Read a string of ASCII data. The `readline` function reads data until it reaches a terminator, removes the terminator, and returns the data.

```
data = readline(u)
```

```
data = 

    "Request Time"
```

Clean Up

When you are finished with the UDP object, clear it and turn off the echo server.

```
clear u
echoudp("off")
```

See Also

`udpport` | `echoudp` | `readline` | `writeline`

More About

- "Write and Read Binary Data Over UDP" on page 8-9

Write and Read Binary Data Over UDP

In this example, write and read binary data with a UDP object.

Write and Read Binary Data with Byte-Type UDP Port

Configure and Connect to the Server

Use an echo server to experiment with the basic functionality of the UDP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echoudp("on",4040)
```

Create a byte-type `udpport` object.

```
u = udpport
```

```
u = 

  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 51595
    NumBytesAvailable: 0

  Show all properties, functions
```

Write Binary Data

Use the `write` function to write the values `1:10` to the server.

```
write(u,1:10,"localhost",4040)
```

The function suspends MATLAB execution until all the data is written or a timeout occurs as specified by the `Timeout` property of the `udpport` object.

By default, the `write` function writes binary data as `uint8` data. For more information about specifying other data types, see `write`.

Read Binary Data

Confirm the success of the write operation by viewing the `NumBytesAvailable` property.

```
u.NumBytesAvailable
```

```
ans = 

    10
```

Since each `uint8` data is one byte and ten values were written, a total of ten bytes are available to be read from the object.

Since the `udpport` object is connected to an echo server, the data you write is returned to the object. Read the first five values of data as `uint8` data.

```
data1 = read(u,5,"uint8")
```

```
data1 =

     1     2     3     4     5
```

After you read data, MATLAB removes it from the buffer. You can use the same command to read the next five bytes of data as `uint8` from the buffer.

```
data2 = read(u,5,"uint8")
```

```
data2 =

     6     7     8     9    10
```

Clean Up

When you are finished with the UDP object, clear it and turn off the echo server.

```
clear u
echoudp("off")
```

Write and Read Binary Data with Datagram-Type UDP Port

Configure and Connect to the Server

Use an echo server to experiment with the basic functionality of the UDP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echoudp("on",4040)
```

Create a datagram-type `udpport` object.

```
u = udpport("datagram")
```

```
u =

  UDPPort with properties:

         IPAddressVersion: "IPV4"
                LocalHost: "0.0.0.0"
                LocalPort: 65390
    NumDatagramsAvailable: 0

  Show all properties, functions
```

Set the maximum number of bytes of data to be written in a datagram packet. This determines the number of datagrams that will be written to the server.

```
u.OutputDatagramSize = 10;
```

Write Binary Data

Use the `write` function to write the values `1:15` to the server.

```
write(u,1:15,"localhost",4040)
```

The function suspends MATLAB execution until all the data is written or a timeout occurs as specified by the `Timeout` property of the `udpport` object.

By default, the `write` function writes binary data as `uint8` data. For more information about specifying other data types, see `write`.

Read Binary Data

Confirm the success of the write operation by viewing the `NumDatagramsAvailable` property.

```
u.NumDatagramsAvailable
```

```
ans =

     2
```

Since you specified each datagram as ten bytes and 15 bytes were written, a total of two datagrams are available to be read from the object.

Since the `udpport` object is connected to an echo server, the data you write is returned to the object. Read the all the data as `uint8` data.

```
data = read(u,u.NumDatagramsAvailable,"uint8")
```

```
data = 

  1×2 Datagram array with properties:

    Data
    SenderAddress
    SenderPort
```

View the first datagram and its values.

```
data1 = data(1)
```

```
data1 = 

  Datagram with properties:

             Data: [1 2 3 4 5 6 7 8 9 10]
    SenderAddress: "127.0.0.1"
       SenderPort: 4040
```

View the second datagram and its values.

```
data2 = data(2)
```

```
data2 = 
```

```
  Datagram with properties:

             Data: [11 12 13 14 15]
    SenderAddress: "127.0.0.1"
       SenderPort: 4040
```

Clean Up

When you are finished with the UDP object, clear it and turn off the echo server.

```
clear u
echoudp("off")
```

See Also

`udpport` | `echoudp` | `read` | `write`

More About

- “Write and Read ASCII Data Over UDP” on page 8-7

Use Callbacks for UDP Communication

You can enhance the power and flexibility of your UDP port by using events and callbacks. An event occurs after a condition is met and can result in one or more callbacks.

While MATLAB is connected to a UDP port, you can use events to display a message, display data, analyze data, and so on. You can control callbacks through callback properties and callback functions. All event types have an associated callback property. Callback functions are MATLAB functions that you write to suit your specific application needs. Execute a callback when a particular event occurs by specifying the name of the callback function as the value for the associated callback property.

Callback Properties

The `udpport` properties and functions associated with callbacks follow.

Property or Function	Purpose
`NumBytesAvailable`	Number of bytes available to read
`BytesAvailableFcn`	Bytes available callback function
`BytesAvailableFcnCount`	Number of bytes of data to trigger callback
`BytesAvailableFcnMode`	Bytes available callback trigger mode
`NumDatagramsAvailable`	Number of datagrams available to read
`DatagramsAvailableFcn`	Datagrams available callback function
`DatagramsAvailableFcnCount`	Number of datagrams to trigger callback
`DatagramsAvailableFcnMode`	Datagrams available callback trigger mode
`configureCallback`	Set callback function and trigger condition for communication with UDP port
`ErrorOccurredFcn`	Callback function triggered by error event
`UserData`	General purpose property for user data

For more information about configuring these properties and functions, see `udpport` Properties on page 24-377.

Use Byte-Mode Callback with Byte-Type UDP Object

This example shows how to set a callback function to trigger when a certain number of bytes of data are available from a byte-type `udpport` object.

1. Create an echo server to connect to from your UDP network socket. An echo server returns the same data that is written to it.

   ```
   echoudp("on",4040)
   ```

2. Create the callback function. Define a callback function `readUDPData` that reads received data and displays it in the command window.

   ```
   function readUDPData(src,~)
       src.UserData = src.UserData + 1;
       disp("Callback Call Count: " + num2str(src.UserData))
   ```

```
    data = read(src,src.BytesAvailableFcnCount,"uint8")
end
```

Save this as an .m file in the directory that you are working in.

3 Create a byte-type `udpport` object `u`.

```
u = udpport("byte","LocalPort",3030)
```

```
u = 

  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 3030
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the `UserData` property to 0.

```
u.UserData = 0;
```

4 Configure callback properties to read and display data each time five bytes are received in MATLAB.

```
configureCallback(u,"byte",5,@readUDPData)
```

5 Write ten bytes of data to trigger the callback function twice.

```
write(u,1:10,"localhost",4040)
```

```
Callback Call Count: 1

data =

     1     2     3     4     5

Callback Call Count: 2

data =

     6     7     8     9    10
```

6 Clear the `udpport` object when you are done working with it. Turn off the echo server.

```
clear u
echoudp("off")
```

Use Terminator-Mode Callback with Byte-Type UDP Object

This example shows how to set a callback function to trigger when a terminator is available to be read from a byte-type `udpport` object.

1 Create an echo server to connect to from your UDP network socket. An echo server returns the same data that is written to it.

```
echoudp("on",4040)
```

2 Create the callback function. Define a callback function `readUDPData` that reads received data and displays it in the command window

```
function readUDPData(src,~)
    src.UserData = src.UserData + 1;
    disp("Callback Call Count: " + num2str(src.UserData))
    data = readline(src)
end
```

Save this as an .m file in the directory that you are working in.

3 Create a byte-type `udpport` object `u`.

```
u = udpport("byte","LocalPort",3030)
```

```
u = 

  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 3030
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the `UserData` property to 0.

```
u.UserData = 0;
```

4 Configure callback properties to read and display data each time ASCII-terminated data is received in MATLAB.

```
configureCallback(u,"terminator",@readUDPData)
```

5 Write a string to trigger the callback function. The `writeline` function automatically appends the terminator to the string before it is written to the server.

```
writeline(u,"hello","localhost",4040)
```

```
Callback Call Count: 1

data = 

    "hello"
```

Write another string to trigger the callback function a second time.

```
writeline(u,"world","localhost",4040)
```

```
Callback Call Count: 2

data = 

    "world"
```

6 Clear the `udpport` object when you are done working with it. Turn off the echo server.

```
clear u
echoudp("off")
```

Use Datagram-Mode Callback with Datagram-Type UDP Object

This example shows how to set a callback function to trigger when a certain number of datagrams are available from a datagram-type `udpport` object.

1. Create an echo server to connect to from your UDP network socket. An echo server returns the same data that is written to it.

   ```
   echoudp("on",4040)
   ```

2. Create the callback function. Define a callback function `readUDPData` that reads received data and displays it in the command window

   ```
   function readUDPData(src,~)
       src.UserData = src.UserData + 1;
       disp("Callback Call Count: " + num2str(src.UserData))
       data = read(src,1,"uint8")
   end
   ```

   Save this as an .m file in the directory that you are working in.

3. Create a datagram-type `udpport` object `u`.

   ```
   u = udpport("datagram","LocalPort",3030)
   ```

   ```
   u = 

     UDPPort with properties:

            IPAddressVersion: "IPV4"
                   LocalHost: "0.0.0.0"
                   LocalPort: 3030
       NumDatagramsAvailable: 0

     Show all properties, functions
   ```

   Set the `UserData` property to 0.

   ```
   u.UserData = 0;
   ```

   Specify the maximum number of bytes to be written in a datagram packet as ten bytes by setting the `OutputDatagramSize` property.

   ```
   u.OutputDatagramSize = 10;
   ```

4. Configure callback properties to read and display data each time one datagram is received in MATLAB.

   ```
   configureCallback(u,"datagram",1,@readUDPData)
   ```

5. Write 30 bytes of data to trigger the callback function three times.

   ```
   write(u,1:30,"localhost",4040)
   ```

   ```
   Callback Call Count: 1

   data = 

     Datagram with properties:
   ```

```
              Data: [1 2 3 4 5 6 7 8 9 10]
     SenderAddress: "127.0.0.1"
        SenderPort: 4040

Callback Call Count: 2

data = 

  Datagram with properties:

              Data: [11 12 13 14 15 16 17 18 19 20]
     SenderAddress: "127.0.0.1"
        SenderPort: 4040

Callback Call Count: 3

data = 

  Datagram with properties:

              Data: [21 22 23 24 25 26 27 28 29 30]
     SenderAddress: "127.0.0.1"
        SenderPort: 4040
```

6 Clear the `udpport` object when you are done working with it. Turn off the echo server.

```
clear u
echoudp("off")
```

See Also

`udpport` | `configureCallback`

More About

- "Communicate Between Two MATLAB Sessions Using User Datagram Protocol" on page 22-118

Transition Your Code to udpport Interface

The udp function, its object functions, and its properties will be removed. Use udpport instead.

udp Interface	udpport Interface	Example
udp	udpport	"Connect to UDP Socket" on page 8-18
fwrite and fread	write and read	"Read and Write" on page 8-18
fprintf	writeline	"Read Terminated String" on page 8-20 "Write and Read Back Data" on page 8-21
fscanf, fgetl, and fgets	readline	"Read Terminated String" on page 8-20 "Read and Parse String Data" on page 8-21 "Write and Read Back Data" on page 8-21
flushinput and flushoutput	flush	"Flush Data from Memory" on page 8-22
Terminator	configureTerminator	"Set Terminator" on page 8-22
BytesAvailableFcnCount, BytesAvailableFcnMode, BytesAvailableFcn, and BytesAvailable	configureCallback	"Set Up Callback Function" on page 8-22
udp Properties	udpport Properties on page 24-377	

Connect to UDP Socket

These examples show how to connect to a UDP socket and disconnect from it using the recommended functionality.

Functionality	Use This Instead
u = udp; fopen(u)	uByte = udpport("byte"); uDatagram = udpport("datagram");
fclose(u) delete(u) clear u	clear uByte clear uDatagram

For more information, see udpport.

Read and Write

These examples use a echo server to show how to perform a binary write and read, and write and read a nonterminated string using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td>

```
echoudp("on",9090)

% u is a udp object
u.DatagramTerminateMode = "off";
fwrite(u,1:5);
data = fread(u,5)


data =

     1
     2
     3
     4
     5
```

</td><td>

```
echoudp("on",9090)

% uByte is a udpport byte object
write(uByte,1:5,"127.0.0.1",9090)
data = read(uByte,5)


data =

     1     2     3     4     5
```

</td></tr>
<tr><td>

```
echoudp("on",9090)

% u is a udp object
u.DatagramTerminateMode = "on";
fwrite(u,1:5);
data = fread(u,1)


data =

     1
     2
     3
     4
     5
```

</td><td>

```
echoudp("on",9090)

% uDatagram is a udpport datagram object
write(uDatagram,1:5)
data = read(uDatagram,1)


data =

  Datagram with properties:

             Data: [1 2 3 4 5]
    SenderAddress: "127.0.0.1"
       SenderPort: 9090
```

</td></tr>
<tr><td>

```
echoudp("on",9090)

% u is a udp object
fwrite(u,"hello","char")
length = 1;
data = fread(u,length,"char")


data =

   104
   101
   108
   108
   111

data = char(data)'


data =

    'hello'
```

</td><td>

```
echoudp("on",9090)

% uByte is a udpport byte object
write(uByte,"hello","string","localhost",9090);
length = 5;
data = read(uByte,length,"string")


data =

    "hello"
```

</td></tr>
</table>

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td></td><td><pre>echoudp("on",9090)

% uDatagram is a udpport datagram object
write(uDatagram,"hello","string","localhost",9090);
length = 1;
data = read(uDatagram,length,"string")

data =

  Datagram with properties:

              Data: "hello"
     SenderAddress: "127.0.0.1"
        SenderPort: 9090</pre></td></tr>
</table>

For more information, see `write` or `read`.

Read Terminated String

This example shows how to perform a terminated string write and read using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td><pre>echoudp("on",9090)

% u is a udp object
u.Terminator = "CR";
fprintf(u,"hello")
data = fscanf(u)

data =

    'hello
     '</pre></td><td rowspan="2"><pre>echoudp("on",9090)

% uByte is a udpport byte object
configureTerminator(uByte,"CR");
writeline(uByte,"hello","127.0.0.1",9090);
data = readline(uByte)

a =

    "hello"</pre></td></tr>
<tr><td><pre>echoudp("on",9090)

% u is a udp object
u.Terminator = "CR";
fprintf(u,"hello")
data = fgetl(u)

data =

    'hello'</pre>
<code>fgetl</code> reads until the specified terminator is reached and then discards the terminator.</td></tr>
</table>

Functionality	Use This Instead

`echoudp("on",9090)`
 	
`% u is a udp object`
`u.Terminator = "CR";`
`fprintf(u,"hello")`
`data = fgets(u)`
 	
`data =`
 	
`    'hello`
`     '`
 	
`fgets` reads until the specified terminator is reached and then returns the terminator.	

For more information, see `writeline` or `readline`.

Read and Parse String Data

This example shows how to read and parse string data using the recommended functionality.

Functionality	Use This Instead

`% u is a udp object`
`data = scanstr(u,';')`
 	
`data =`
 	
`  3×1 cell array`
 	
`    {'a'}`
`    {'b'}`
`    {'c'}`
	`% uByte is a udpport byte object`
	`data = readline(uByte)`
	
	`data =`
	
	`    "a;b;c"`
	
	`data = strsplit(data,";")`
	
	`data =`
	
	`  1×3 string array`
	
	`    "a"    "b"    "c"`

For more information, see `readline`.

Write and Read Back Data

This example shows how to write ASCII terminated data and read ASCII terminated data back using the recommended functionality.

Functionality	Use This Instead

`% u is a udp object`
`data = query(u,'ctrlcmd')`
 	
`data =`
 	
`    'success'`
	`% uByte is a udpport byte object`
	`writeline(uByte,"ctrlcmd")`
	`data = readline(uByte)`
	
	`data =`
	
	`    "success"`

For more information, see `writeline` or `readline`.

Flush Data from Memory

This example shows how to flush data from the buffer using the recommended functionality.

Functionality	Use This Instead
`% u is a udp object` `flushinput(u)`	`% u is a udpport byte or datagram object` `flush(u,"input")`
`% u is a udp object` `flushoutput(u)`	`% u is a udpport byte or datagram object` `flush(u,"output")`

`% u is a udp object`
`flushinput(u)`
`flushoutput(u)`
	`% u is a udpport byte or datagram object`
	`flush(u)`

For more information, see `flush`.

Set Terminator

These examples show how to set the terminator using the recommended functionality.

Functionality	Use This Instead
`% u is a udp object` `u.Terminator = "CR/LF";`	`% u is a udpport byte or datagram object` `configureTerminator(u,"CR/LF")`
`% u is a udp object` `u.Terminator = {"CR/LF" [10]};`	`% u is a udpport byte or datagram object` `configureTerminator(u,"CR/LF",10)`

For more information, see `configureTerminator`.

Set Up Callback Function

These examples show how to set up a callback function using the recommended functionality.

<table>
<tr><th>Functionality</th><th>Use This Instead</th></tr>
<tr><td rowspan="2"><pre>
% u is a udp object
u.BytesAvailableFcnCount = 5
u.BytesAvailableFcnMode = "byte"
u.BytesAvailableFcn = @mycallback

function mycallback(src,evt)
   data = fread(src,src.BytesAvailableFcnCount);
   disp(evt)
   disp(evt.Data)
end

    Type: 'BytesAvailable'
    Data: [1×1 struct]

    AbsTime: [2019 12 21 16 35 9.7032]
</pre></td><td><pre>
% uByte is a udpport byte object
configureCallback(uByte,5,@mycallback);

function mycallback(src,evt)
   data = read(src,src.BytesAvailableFcnCount);
   disp(evt)
end

  ByteAvailableInfo with properties:

    BytesAvailableFcnCount: 5
                   AbsTime: 21-Dec-2019 12:23:01
</pre></td></tr>
<tr><td><pre>
% uDatagram is a udpport datagram object
configureCallback(uDatagram,"datagram",1,@mycallback)

function mycallback(src,evt)
   data = read(src,src.DatagramsAvailableFcnCount);
   disp(evt)
end

  DatagramAvailableInfo with properties:

    DatagramsAvailableFcnCount: 1
                       AbsTime: 21-Dec-2019 12:23:01
</pre></td></tr>
<tr><td><pre>
% u is a udp object
u.Terminator = "CR"
u.BytesAvailableFcnMode = "terminator"
u.BytesAvailableFcn = @mycallback

function mycallback(src,evt)
   data = fscanf(src);
   disp(evt)
   disp(evt.Data)
end

    Type: 'BytesAvailable'
    Data: [1×1 struct]

    AbsTime: [2019 12 21 16 35 9.7032]
</pre></td><td><pre>
% uByte is a udpport byte object
configureCallback(uByte,"terminator",@mycallback);

function mycallback(src,evt)
   data = readline(src);
   disp(evt)
end

  TerminatorAvailableInfo with properties:

                   AbsTime: 21-Dec-2019 12:23:01
</pre></td></tr>
</table>

For more information, see `configureCallback`.

See Also

`udpport`

More About

- R2020a TCP/IP and UDP Interface Topics

Basic Workflow to Read and Write Data over UDP

This example shows the basic workflow of text read and write operations with a UDP object connected to a remote instrument.

The instrument used is an echo server on a Linux-based PC. An echo server is a service available from the operating system that returns (echoes) received data to the sender. The host name is `daqlab11` and the port number is `7`. The host name is assigned by your network administrator.

1. **Create an instrument object** — Create a UDP object associated with `daqlab11`.

   ```
   u = udp('daqlab11',7);
   ```
2. **Write and read data** — You use the `fprintf` function to write text data to the instrument. For example, write the following string to the echo server.

   ```
   fprintf(u,'Request Time')
   ```

   UDP sends and receives data in blocks that are called datagrams. Each time you write or read data with a UDP object, you are writing or reading a datagram. For example, the string sent to the echo server constitutes a datagram with 13 bytes — 12 ASCII bytes plus the line feed terminator.

   You use the `fscanf` function to read text data from the echo server.

   ```
   fscanf(u)
   ans =
   Request Time
   ```

   The `DatagramTerminateMode` property indicates whether a read operation terminates when a datagram is received. By default, `DatagramTerminateMode` is `on` and a read operation terminates when a datagram is received. To return multiple datagrams in one read operation, set `DatagramTerminateMode` to `off`.

   The following commands write two datagrams. Note that only the second datagram sends the terminator character.

   ```
   fprintf(u,'%s','Request Time')
   fprintf(u,'%s\n','Request Time')
   ```

   Since `DatagramTerminateMode` is `off`, `fscanf` reads across datagram boundaries until the terminator character is received.

   ```
   u.DatagramTerminateMode = 'off'
   data = fscanf(u)
   data =
   Request TimeRequest Time
   ```
3. **Disconnect and clean up** — When you no longer need `u`, you should disconnect it from the host, and remove it from memory and from the MATLAB workspace.

   ```
   fclose(u)
   delete(u)
   clear u
   ```

Note UDP ports can be shared by other applications to allow for multiple applications to listen to the UDP datagrams on that port. This allows for the ability to listen to UDP broadcasts on the same local

port number in both MATLAB and other applications. You can enable and disable this capability with a new property of the UDP object called `EnablePortSharing`. See “Enable Port Sharing over UDP” on page 8-4.

Asynchronous Read and Write Operations over UDP

This section provides details and examples exploring asynchronous read and write operations with a UDP object.

In this section...
“Functions and Properties” on page 8-26
“Synchronous Versus Asynchronous Operations” on page 8-26
“Configuring and Connecting to the Server” on page 8-27
“Reading Data Asynchronously” on page 8-27
“Reading Data Asynchronously Using Continuous ReadAsyncMode” on page 8-28
“Reading Data Asynchronously Using Manual ReadAsyncMode” on page 8-28
“Defining an Asynchronous Read Callback” on page 8-29
“Using Callbacks During an Asynchronous Read” on page 8-29
“Writing Data Asynchronously” on page 8-30
“Cleanup” on page 8-30

Functions and Properties

These functions are associated with reading and writing text asynchronously:

Function	Purpose
`fprintf`	Write text to a server.
`readasync`	Asynchronously read bytes from a server.
`stopasync`	Stop an asynchronous read or write operation.

These properties are associated with reading and writing text asynchronously:

Property	Purpose
`BytesAvailable`	Indicates the number of bytes available in the input buffer.
`TransferStatus`	Indicates what type of asynchronous operation is in progress.
`ReadAsyncMode`	Indicates whether data is read continuously in the background or whether you must call the `readasync` function to read data asynchronously.

Additionally, you can use all the callback properties during asynchronous read and write operations.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Synchronous Versus Asynchronous Operations

The object can operate in synchronous mode or in asynchronous mode. When the object is operating synchronously, the read and write routines block the MATLAB command line until the operation has

completed or a timeout occurs. When the object is operating asynchronously, the read and write routines return control immediately to the MATLAB command line.

Additionally, you can use callback properties and callback functions to perform tasks as data is being written or read. For example, you can create a callback function that notifies you when the read or write operation has finished.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the UDP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echoudp('on', 8000);
```

You need to create a UDP object. In this example, create a UDP object associated with the host 127.0.0.1 (your local machine), port 8000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
u = udp('127.0.0.1', 8000);
```

Before you can perform a read or write operation, you must connect the UDP object to the server with the `fopen` function.

```
fopen(u)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
u.Status
ans =
    open
```

Note UDP ports can be shared by other applications to allow for multiple applications to listen to the UDP datagrams on that port. This allows for the ability to listen to UDP broadcasts on the same local port number in both MATLAB and other applications. You can enable and disable this capability with a new property of the UDP object called `EnablePortSharing`. See “Enable Port Sharing over UDP” on page 8-4.

Reading Data Asynchronously

You can read data asynchronously with the UDP object in one of these two ways:

- Continuously, by setting `ReadAsyncMode` to `continuous`. In this mode, data is automatically stored in the input buffer as it becomes available from the server.
- Manually, by setting `ReadAsyncMode` to `manual`. In this mode, you must call the `readasync` function to store data in the input buffer.

The `fscanf`, `fread`, `fgetl` and `fgets` functions are used to bring the data from the input buffer into MATLAB. These functions operate synchronously.

Reading Data Asynchronously Using Continuous ReadAsyncMode

To read data continuously:

```
u.ReadAsyncMode = 'continuous';
```

To send a string to the echoserver:

```
fprintf(u, 'Hello net.');
```

Because the `ReadAsyncMode` property is set to `continuous`, the object is continuously asking the server if any data is available. The echoserver sends data as soon as it receives data. The data is then read from the server and is stored in the object's input buffer.

```
u.BytesAvailable
ans =
    11
```

You can bring the data from the object's input buffer into the MATLAB workspace with `fscanf`.

```
mystring = fscanf(u)
mystring =
    Hello net.
```

Reading Data Asynchronously Using Manual ReadAsyncMode

You can also read data manually.

```
u.ReadAsyncMode = manual;
```

Now, send a string to the echoserver.

```
fprintf(u, 'Hello net.');
```

Once the last `fprintf` function completes, the server begins sending data. However, because `ReadAsyncMode` is set to `manual`, the object is not reading the data being sent from the server. Therefore no data is being read and placed in the input buffer.

```
u.BytesAvailable
ans =
    0
```

The `readasync` function can asynchronously read the data from the server. The `readasync` function returns control to the MATLAB command line immediately.

The `readasync` function takes two input arguments. The first argument is the server object and the second argument is the `size`, the amount of data to be read from the server.

The `readasync` function without a `size` specified assumes `size` is given by the difference between the `InputBufferSize` property value and the `BytesAvailable` property value. The asynchronous read terminates when:

- The terminator is read as specified by the `Terminator` property
- The specified number of bytes have been read
- A timeout occurs as specified by the `Timeout` property

- The input buffer is filled

An error event will be generated if readasync terminates due to a timeout.

The object starts querying the server for data when the readasync function is called. Because all the data was sent before the readasync function call, no data will be stored in the input buffer and the data is lost.

When the UDP object is in manual mode (the ReadAsyncMode property is configured to manual), data that is sent from the server to the computer is not automatically stored in the input buffer of the UDP object. Data is not stored until readasync or one of the blocking read functions is called.

Manual mode should be used when a stream of data is being sent from your server and you only want to capture portions of the data.

Defining an Asynchronous Read Callback

You can configure a UDP object to notify you when a terminator has been read using the dispcallback function.

```
u.ReadAsyncMode = 'continuous';
u.BytesAvailableFcn = 'dispcallback';
```

Note, the default value for the BytesAvailableFcnMode property indicates that the callback function defined by the BytesAvailableFcn property will be executed when the terminator has been read.

```
u.BytesAvailableFcnMode
ans =
    terminator
```

The callback function dispcallback displays event information for the specified event. Using the syntax dispcallback(obj, event), it displays a message containing the type of event, the name of the object that caused the event to occur, and the time the event occurred.

```
callbackTime = datestr(datenum(event.Data.AbsTime));
fprintf(['A ' event.Type ' event occurred for ' obj.Name ' at '
     callbackTime '.\n']);
```

Using Callbacks During an Asynchronous Read

Once the terminator is read from the server and placed in the input buffer, dispcallback is executed and a message is posted to the MATLAB command window indicating that a BytesAvailable event occurred.

```
fprintf(u, 'Hello net.')
u.BytesAvailable
ans =
    11

data = fscanf(u)
data =
    Hello net.
```

If you need to stop an asynchronous read or write operation, you do not have to wait for the operation to complete. You can use the stopasync function to stop the asynchronous read or write.

```
stopasync(u);
```

The data that has been read from the server remains in the input buffer. You can use one of the synchronous read functions to bring this data into the MATLAB workspace. However, because this data represents the wrong data, the `flushinput` function is called to remove all data from the input buffer.

```
flushinput(u);
```

Writing Data Asynchronously

You can perform an asynchronous write with the `fprintf` or `fwrite` functions by passing `'async'` as the last input argument.

Configure the object to notify you when an asynchronous write operation completes.

```
u.OutputEmptyFcn = 'dispcallback';
fprintf(u, 'Hello net.', 'async')
```

UDP sends and receives data in blocks that are called datagrams. Each time you write or read data with a UDP object, you are writing or reading a datagram. In the example below, a datagram with 11 bytes (10 ASCII bytes plus the LF terminator) will be sent to the echoserver. Then the echoserver will send back a datagram containing the same 11 bytes.

Configure the object to notify you when a datagram has been received.

```
u.DatagramReceivedFcn = 'dispcallback';
fprintf(u, 'Hello net.', 'async')
```

Note If you need to stop an asynchronous read or write operation, you do not have to wait for the operation to complete. You can use the `stopasync` function to stop the asynchronous read or write.

Cleanup

If you are finished with the UDP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(u);
delete(u);
clear u
```

```
echoudp('off');
```

Controlling Instruments Using I2C

- “I2C Interface Overview” on page 9-2
- “Configuring I2C Communication” on page 9-3
- “Transmitting Data Over the I2C Interface” on page 9-6
- “I2C Interface Usage Requirements and Guidelines” on page 9-10

I2C Interface Overview

I2C Communication

I2C, or Inter-Integrated Circuit, is a chip-to-chip protocol supporting two-wire communication. An `i2c` object represents a connection between MATLAB and an I2C adapter board. The adapter has one or more sensor chips connected to it. MATLAB sends commands to the adapter board, which is the I2C controller device, in order to communicate with the chip, which is the I2C peripheral device. MATLAB always has the role of I2C controller and cannot be used in the peripheral role.

Supported adapters are the Total Phase Aardvark I2C/SPI Host Adapter and the National Instruments USB-845x adapter board. Some applications of this interface include communication with SPD EEPROM and NVRAM chips, communication with SMBus devices, controlling accelerometers, accessing low-speed DACs and ADCs, changing settings on color monitors using the display data channel, changing sound volume in intelligent speakers, reading hardware monitors and diagnostic sensors, visualizing data sent from an I2C sensor, and turning on or off the power supply of system components.

Use `fread` and `fwrite` to communicate with the chip. Identify I2C devices using `instrhwinfo('i2c')`.

Supported Platforms for I2C

You need to have either a Total Phase Aardvark host adapter or a NI USB-845x adapter board installed to use the `i2c` interface. The following sections contain the supported platforms for each option.

Using Total Phase Aardvark

The I2C interface is supported on these platforms when used with the Aardvark host adapter:

- Linux – The software works with Red Hat Enterprise Linux 4 and 5 with kernel 2.6. It may also be successful with SuSE and Ubuntu distributions.
- Microsoft Windows 64-bit

Note For R2018b and R2018a, you cannot use the Aardvark adapter for I2C or SPI interfaces on the macOS platform. You can still use it on Windows and Linux. For releases prior to R2018a, you can use it on all three platforms, including macOS.

Using NI USB-845x

The I2C interface is supported on these platforms when used with the NI USB-845x host adapter:

- Microsoft Windows 64-bit

See Also

`i2c`

Configuring I2C Communication

You need to have either a Total Phase Aardvark host adapter or a NI USB-845x adapter board installed to use the `i2c` interface. The following sections describe configuration for each option.

Configuring Total Phase Aardvark

To use the I2C interface with the Aardvark adapter, you must download the Hardware Support Package to obtain the necessary files. You must also download the USB device driver from the vendor.

If you do not have the Aardvark driver installed, see “Install the Total Phase Aardvark I2C/SPI Interface Support Package” on page 15-9.

The aardvark.dll file that comes with the Total Phase Aardvark adapter board must be available in one of the following locations for use on Windows platforms.

- The location where MATLAB was started from (Bin folder).
- The MATLAB current folder (PWD).
- The Windows folder `C:\winnt` or `C:\windows`.
- The folders listed in the PATH environment variable.

Ensure that the Aardvark adapter is installed properly.

```
instrhwinfo('i2c')
```

```
ans = 

  HardwareInfo with properties:

    InstalledAdaptors: {'Aardvark'  'NI845x'}
       JarFileVersion: 'Version 4.1'

Access to your hardware may be provided by a support package. Go to the Support Package Installe
```

Look at the adapter properties.

```
instrhwinfo('i2c','Aardvark')
```

```
ans = 

  HardwareInfo with properties:

             AdaptorDllName: 'C:\Program Files\MATLAB\R2019b\toolbox\instrument\instrumentadapto
          AdaptorDllVersion: 'Version 4.1'
                AdaptorName: 'Aardvark'
              BoardIdsInUse: [1×0 double]
          InstalledBoardIDs: 0
       DetectedBoardSerials: {'2237482577 (BoardIndex: 0)'}
      ObjectConstructorName: 'i2c('Aardvark', BoardIndex, RemoteAddress);'
              VendorDllName: 'aardvark.dll'
    VendorDriverDescription: 'Total Phase I2C Driver'

Access to your hardware may be provided by a support package. Go to the Support Package Installe
```

You can create an I2C object using the `i2c` function. The example in the next section uses an I2C object called `eeprom` that communicates to an EEPROM chip. View the properties after creating the object.

```
eeprom = i2c('aardvark',0,'50h')
```

```
   I2C Object : I2C-0-50h

   Communication Settings
      BoardIndex          0
      BoardSerial         2237482577
      BitRate:            100 kHz
      RemoteAddress:      50h
      Vendor:             aardvark


   Communication State
      Status:             closed
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
```

You can see that the communication settings properties reflect what was used to create the object – `BoardIndex` of `0` and `RemoteAddress` of `50h`. For information about other properties, see "Properties" on page 24-182.

Configuring NI USB-845x

To use the I2C interface with the NI USB-845x adapter, you must download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the NI USB-845x driver installed, see "Install the NI-845x I2C/SPI Interface Support Package" on page 15-8.

Ensure that the NI USB-845x adapter is installed properly.

```
instrhwinfo('i2c')
```

```
ans = 

  HardwareInfo with properties:

    InstalledAdaptors: {'Aardvark'  'NI845x'}
       JarFileVersion: 'Version 4.1'

Access to your hardware may be provided by a support package. Go to the Support Package Installer
```

Look at the NI USB-845x adapter properties.

```
instrhwinfo('i2c','NI845x')
```

```
ans = 
```

```
  HardwareInfo with properties:

             AdaptorDllName: 'C:\Program Files\MATLAB\R2019b\toolbox\instrument\instrumentadapto
          AdaptorDllVersion: 'Version 4.1'
                AdaptorName: 'NI845x'
              BoardIdsInUse: [1×0 double]
          InstalledBoardIDs: [1×0 double]
       DetectedBoardSerials: {0×1 cell}
      ObjectConstructorName: 'i2c('NI845x', BoardIndex, RemoteAddress);'
              VendorDllName: 'Ni845x.dll'
    VendorDriverDescription: 'National Instruments NI USB 845x Driver'

Access to your hardware may be provided by a support package. Go to the Support Package Installe
```

You can create an I2C object using the `i2c` function. View the properties after creating the object.

```
i2cobj = i2c('NI845x',0,'10h')
```

```
   I2C Object : I2C-0-10h

   Communication Settings
      BoardIndex          0
      BoardSerial         0
      BitRate:            100 kHz
      RemoteAddress:      10h
      Vendor:             NI845x


   Communication State
      Status:             closed
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
```

You can see that the communication settings properties reflect what was used to create the object – `BoardIndex` of `0` and `RemoteAddress` of `10h`. For information about other properties, see “Properties” on page 24-182.

See Also

`i2c`

More About

- “Install the Total Phase Aardvark I2C/SPI Interface Support Package” on page 15-9
- “Install the NI-845x I2C/SPI Interface Support Package” on page 15-8

Transmitting Data Over the I2C Interface

The typical workflow involves adapter discovery, connection, communication, and cleanup. Discovery can be done only at the adapter level. You need to have either a Total Phase Aardvark host adapter or a NI USB-845x adapter board installed to use the `i2c` interface.

Aardvark Example

This example shows how to communicate with an EEPROM chip on a circuit board, with an address of 50 hex and a board index of 0, using the Aardvark adapter.

To communicate with an EEPROM chip:

1 Ensure that the Aardvark adapter is installed so that you can use the `i2c` interface.

```
instrhwinfo('i2c')


ans = 

  HardwareInfo with properties:

    InstalledAdaptors: {'Aardvark'  'NI845x'}
       JarFileVersion: 'Version 4.1'

Access to your hardware may be provided by a support package. Go to the Support Package Insta
```

2 Look at the adapter properties.

```
instrhwinfo('i2c','Aardvark')


ans = 

  HardwareInfo with properties:

               AdaptorDllName: 'C:\Program Files\MATLAB\R2019b\toolbox\instrument\instrumentada
            AdaptorDllVersion: 'Version 4.1'
                  AdaptorName: 'Aardvark'
                BoardIdsInUse: [1×0 double]
            InstalledBoardIDs: 0
         DetectedBoardSerials: {'2237482577 (BoardIndex: 0)'}
        ObjectConstructorName: 'i2c('Aardvark', BoardIndex, RemoteAddress);'
                VendorDllName: 'aardvark.dll'
      VendorDriverDescription: 'Total Phase I2C Driver'

Access to your hardware may be provided by a support package. Go to the Support Package Insta
```

Make sure that you have the Aardvark software driver installed and that the aardvark.dll is on your MATLAB path. For details, see “I2C Interface Usage Requirements and Guidelines” on page 9-10.

3 Create the I2C object called `eeprom`, using the following properties.

```
% Vendor = aardvark
% BoardIndex = 0
% RemoteAddress = 50h
```

```
eeprom = i2c('aardvark',0,'50h')

   I2C Object : I2C-0-50h

   Communication Settings
      BoardIndex          0
      BoardSerial         2237482577
      BitRate:            100 kHz
      RemoteAddress:      50h
      Vendor:             aardvark


   Communication State
      Status:             closed
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
```

You must provide these three inputs to create the object. To identify the address of the chip, consult its documentation or data sheet. You can also find the address by scanning for instruments in the Test & Measurement Tool. In the tool, right-click the **I2C** node and select **Scan for I2C adaptors**. Any chips found by the scan is listed in the hardware tree. The listing includes the remote address of the chip.

4. Connect to the chip.

   ```
   fopen(eeprom)
   ```

5. Write `'Hello World!'` to the EEPROM chip. Data is written page-by-page in I2C. Each page contains eight bytes. The page address needs to be mentioned before every byte of data written.

   The first byte of the string `'Hello World!'` is `'Hello Wo'`. Its page address is 0.

   ```
   fwrite(eeprom,[0 'Hello Wo'])
   ```

   The second byte of the string `'Hello World!'` is `'rld!'`. Its page address is 8.

   ```
   fwrite(eeprom,[8 'rld!'])
   ```

6. Read data back from the chip using the `fread` function. The chip returns the characters sent to it.

   To start reading from the first byte of the first page, write a zero to the `i2c` object.

   ```
   fwrite(eeprom,0)
   ```

   ```
   char(fread(eeprom,12))'
   ```

   ```
   ans =

       'Hello World!'
   ```

7. Disconnect the I2C device and clear the object from the workspace.

   ```
   fclose(eeprom)
   clear eeprom
   ```

NI USB-845x Example

This example shows how to communicate with an Analog Devices® ADXL345 sensor chip on a circuit board, using an address of 53 hex and a board index of 0 on a NI USB-845x adapter. In this case, the NI USB-845x adapter board is plugged into the computer (via the USB port), and a circuit board containing the sensor chip is connected to the host adapter board via wires. Note that the circuit has external pullups, as the NI USB-8451 adapter used in this example does not have internal pullups.

To communicate with a sensor chip:

1. Ensure that the NI USB-845x adapter is installed so that you can use the `i2c` interface.

```
instrhwinfo('i2c')


ans = 

  HardwareInfo with properties:

    InstalledAdaptors: {'Aardvark'  'NI845x'}
       JarFileVersion: 'Version 4.1'

Access to your hardware may be provided by a support package. Go to the Support Package Insta
```

2. Look at the NI USB-845x adapter properties.

```
instrhwinfo('i2c','NI845x')


ans = 

  HardwareInfo with properties:

               AdaptorDllName: 'C:\Program Files\MATLAB\R2019b\toolbox\instrument\instrumentada
            AdaptorDllVersion: 'Version 4.1'
                  AdaptorName: 'NI845x'
                BoardIdsInUse: [1×0 double]
            InstalledBoardIDs: [1×0 double]
         DetectedBoardSerials: {0×1 cell}
        ObjectConstructorName: 'i2c('NI845x', BoardIndex, RemoteAddress);'
                VendorDllName: 'Ni845x.dll'
      VendorDriverDescription: 'National Instruments NI USB 845x Driver'

Access to your hardware may be provided by a support package. Go to the Support Package Insta
```

   Make sure that you have the NI USB-845x software driver installed. For details, see “I2C Interface Usage Requirements and Guidelines” on page 9-10.

3. Create the I2C object called `i2cobj`, using these properties:

```
% Vendor = NI845x
% BoardIndex = 0
% RemoteAddress = 53h

i2cobj = i2c('NI845x',0,'53h');
```

   You must provide these three inputs to create the object. To identify the address of the chip, consult its documentation or data sheet. You can also find the address by scanning for instruments in the Test & Measurement Tool. In the tool, right-click the **I2C** node and select

Scan for I2C adaptors. Any chips found by the scan is listed in the hardware tree. The listing includes the remote address of the chip.

4. Connect to the chip.

```
fopen(i2cobj)
```

5. Write to the sensor chip. Read the documentation or data sheet of the chip in order to know what the remote address is and other information about the chip. Usually chip manufacturers provide separate read and write addresses. The adapter boards only take one address (the read address) and handle conversions to read and write addresses.

   In this case, the chip's device ID register is at address `0`, so you need to write a 0 to the chip indicating you would like to read or write to the register.

```
fwrite(i2cobj,0)
```

6. Read data back from the chip using the `fread` function. By sending one byte, you can read back the device ID registry. For this chip, the read-only device ID registry is 229.

```
fread(i2cobj,1)

ans =

   229
```

7. Disconnect the I2C device and clear the object from the workspace.

```
fclose(i2cobj)
clear i2cobj
```

See Also

`i2c` | `fopen` | `fread` | `fwrite` | `fclose`

More About

- "I2C Interface Usage Requirements and Guidelines" on page 9-10

I2C Interface Usage Requirements and Guidelines

The I2C interface does not support asynchronous behavior. Therefore, functions such as `fprintf`, `fscanf`, and `query` do not work. Use `fread` and `fwrite` to communicate using this interface.

You need to have either a Total Phase Aardvark host adapter or a NI USB-845x adapter board installed to use the `i2c` interface. The following sections describe requirements for each option.

Aardvark-specific Requirements

To use the I2C interface with the Aardvark adapter, you must download the Hardware Support Package to obtain the necessary files. You must also download the USB device driver from the vendor.

If you do not have the Aardvark driver installed, see “Install the Total Phase Aardvark I2C/SPI Interface Support Package” on page 15-9.

You must install the Aardvark Software API and Share Library appropriate for your operating system.

The aardvark.dll file that comes with the Total Phase Aardvark adapter board must be available in one of the following locations for use on Windows platforms.

- The location where MATLAB was started from (Bin folder).
- The MATLAB current folder (PWD).
- The Windows folder `C:\winnt` or `C:\windows`.
- The folders listed in the PATH environment variable.

The aardvark.so file that comes with the Total Phase Aardvark adapter board must be in your MATLAB path for use on Linux platforms.

If you repower your Aardvark board, set the GPIO pins to output to get communication with a device to work. By default they are configured as input.

NI USB-845x-specific Requirements

To use the I2C interface with the NI USB-845x adapter, you must download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the NI USB-845x driver installed, see “Install the NI-845x I2C/SPI Interface Support Package” on page 15-8.

Devices differ in their use of pullups. The Aardvark and NI USB-8452 adapters have internal pullup resistors to tie both bus lines to VDD, and can be set programmatically. The NI USB-8451 adapter does not have this type of internal pullup resistor and, therefore, requires external pullups. Consult your device documentation to ensure that you are using the correct pullups.

See Also

More About

- “Install the Total Phase Aardvark I2C/SPI Interface Support Package” on page 15-9
- “Install the NI-845x I2C/SPI Interface Support Package” on page 15-8

Controlling Instruments Using SPI

- "SPI Interface Overview" on page 10-2
- "Configuring SPI Communication" on page 10-3
- "Transmitting Data Over the SPI Interface" on page 10-7
- "Using Properties on the SPI Object" on page 10-13
- "SPI Interface Usage Requirements and Guidelines" on page 10-16

SPI Interface Overview

In this section...

"SPI Communication" on page 10-2
"Supported Platforms for SPI" on page 10-2

SPI Communication

SPI, or Serial Peripheral Interface, is a synchronous serial data link standard that operates in full duplex mode. It is commonly used in the test and measurement field. Typical uses include communicating with micro controllers, EEPROMs, A2D devices, embedded controllers, etc.

Instrument Control Toolbox SPI support lets you open connections with individual chips and to read and write over the connections to individual chips using an Aardvark or NI-845x host adaptor.

The primary uses for the `spi` interface involve the `write`, `read`, and `writeAndRead` functions for synchronously reading and writing binary data. To identify SPI devices in Instrument Control Toolbox, use the `instrhwinfo` function on the SPI interface, called `spi`.

Supported Platforms for SPI

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface. The following sections contain the supported platforms for each option.

Using Aardvark

The SPI interface is supported on these platforms when used with the Aardvark host adaptor:

- Linux — Red Hat® Enterprise Linux 4 and 5 with kernel 2.6, and possibly SUSE® and Ubuntu distributions.
- Microsoft Windows 64-bit

Note For R2018b and R2018a, you cannot use the Aardvark adaptor for I2C or SPI interfaces on the macOS platform. You can still use it on Windows and Linux. For releases prior to R2018a, you can use it on all three platforms, including the Mac.

Using NI-845x

The SPI interface is supported on these platforms when used with the NI-845x host adaptor:

- Microsoft Windows 64-bit

Configuring SPI Communication

You must have a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface.

Configuring Aardvark

To use the SPI interface with the Aardvark adaptor, you must download the Hardware Support Package to obtain the necessary files. You must also download the USB device driver from the vendor.

If you do not have the Aardvark driver installed, see "Install the Total Phase Aardvark I2C/SPI Interface Support Package" on page 15-9.

The `aardvark.dll` file that comes with the Total Phase Aardvark adaptor board must be available in one of these locations for use on Windows platforms:

- The location where MATLAB was started from (bin folder)
- The MATLAB current folder (PWD)
- The Windows folder `C:\winnt` or `C:\windows`
- The folders listed in the path environment variable

Ensure that the Aardvark adaptor is installed properly.

```
>> instrhwinfo('spi')

ans =

    SupportedVendors: {'aardvark'}
    InstalledVendors: {'aardvark'}
```

Look at the adaptor properties.

```
>> instrhwinfo('spi' , 'Aardvark')

ans =

            VendorName: 'aardvark'
     VendorDescription: 'Total Phase I2C/SPI Driver'
     VendorLibraryName: 'aardvark.dll'
     InstalledBoardIds: {[0]}
    BoardSerialNumbers: {'2237722838'}
    ObjectConstructors: {'spi('aardvark', 0, 0)'}
```

Create a SPI object using the `spi` function. This example uses a SPI object called `S` that communicates to an EEPROM chip. Create the object using the `BoardIndex` and `Port` numbers, which are `0` in both cases.

```
S = spi('aardvark', 0, 0);
```

Display the object properties.

```
>> disp(S)
SPI Object :

    Adapter Settings
        BoardIndex:             0
        BoardSerial:            2237722838
        VendorName:             aardvark

    Communication Settings
        BitRate:                1000000 Hz
        ChipSelect:             0
        ClockPhase:             FirstEdge
        ClockPolarity:          IdleLow
        Port:                   0

    Communication State
        ConnectionStatus:       Disconnected

    Read/Write State
        TransferStatus:         Idle
```

The communication settings properties reflect what was used to create the object – `BoardIndex` of `0` and `Port` of `0`. For information about other properties, see “Using Properties on the SPI Object” on page 10-13.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Configuring NI-845x

To use the SPI interface with the NI-845x adaptor, download the hardware support package to obtain the latest driver, if you do not already have the driver installed.

If you do not have the NI-845x driver installed, see “Install the NI-845x I2C/SPI Interface Support Package” on page 15-8 to install it.

Ensure that the NI-845x adaptor is installed properly.

```
>> instrhwinfo('spi')

ans =

  HardwareInfo with properties:

    SupportedVendors: {'aardvark'  'ni845x'}
    InstalledVendors: {'ni845x'}
```

Look at the NI-845x adaptor properties.

```
>> instrhwinfo('spi', 'ni845x')

ans =

  HardwareInfo with properties:

            VendorName: 'ni845x'
     VendorDescription: 'National Instruments'
     VendorLibraryName: 'Ni845x.dll'
     InstalledBoardIds: {[0]}
    BoardSerialNumbers: {'0183388B'}
    ObjectConstructors: {'spi('ni845x', 0, 0)'}
```

Create a SPI object using the `spi` function. The example in the next section uses a SPI object called `s2` that communicates with an EEPROM chip. Create the object using the `BoardIndex` and `Port` numbers, which are `0` in both cases.

```
s2 = spi('ni845x', 0, 0);
```

Display the object properties.

```
>> disp(s2)
SPI Object :

    Adapter Settings
        BoardIndex:            0
        BoardSerial:           0183388B
        VendorName:            ni845x

    Communication Settings
        BitRate:               1000000 Hz
        ChipSelect:            0
        ClockPhase:            FirstEdge
        ClockPolarity:         IdleLow
        Port:                  0

    Communication State
        ConnectionStatus:      Disconnected

    Read/Write State
        TransferStatus:        Idle
```

The communication settings properties reflect what was used to create the object – `BoardIndex` of `0` and `Port` of `0`. For information about other properties, see "Using Properties on the SPI Object" on page 10-13.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Transmitting Data Over the SPI Interface

The typical workflow for transmitting data over the SPI interface involves adaptor discovery, connection, communication, and cleanup. Discovery can be done only at the adaptor level. You must have a Total Phase Aardvark adaptor or an NI-845x adaptor board installed to use the `spi` interface.

Transmit Data Over SPI Using Aardvark

This example shows how to communicate with an EEPROM chip on a circuit board, with a board index of 0 and using port 0.

1. Ensure that the Aardvark adaptor is installed so that you can use the `spi` interface.

```
>> instrhwinfo('spi')

ans =

    SupportedVendors: {'aardvark'}
    InstalledVendors: {'aardvark'}
```

2. Look at the adaptor properties.

```
>> instrhwinfo('spi' , 'Aardvark')

ans =

           VendorName: 'aardvark'
    VendorDescription: 'Total Phase I2C/SPI Driver'
    VendorLibraryName: 'aardvark.dll'
    InstalledBoardIds: {[0]}
   BoardSerialNumbers: {'2237722838'}
   ObjectConstructors: {'spi('aardvark', 0, 0)'}
```

   Make sure that you have the Aardvark software driver installed and that the `aardvark.dll` is on your MATLAB path. For details, see “SPI Interface Usage Requirements and Guidelines” on page 10-16.

3. Create the SPI object called S, using these properties:

```
% Vendor = aardvark
% BoardIndex = 0
% Port = 0

S = spi('aardvark', 0, 0);
```

   You must provide these three parameters to create the object.

4. Look at the object properties.

```
>> S

S =

SPI Object :

    Adapter Settings
        BoardIndex:             0
        BoardSerial:            2237722838
        VendorName:             aardvark

    Communication Settings
        BitRate:                1000000 Hz
        ChipSelect:             0
        ClockPhase:             FirstEdge
        ClockPolarity:          IdleLow
        Port:                   0

    Communication State
        ConnectionStatus:       Disconnected

    Read/Write State
        TransferStatus:         Idle
```

When you create the `spi` object, default communication settings are used, as shown here. To change any of these settings, see "Using Properties on the SPI Object" on page 10-13 for more information and a list of the properties.

5. Connect to the chip.

```
connect(S);
```

6. Read and write to the chip.

```
% Create a variable containing the data to write
dataToWrite = [3 0 0 0];

% Write the binary data to the chip
write(S, dataToWrite);

% Create a variable that contains the number of values to read
numData = 5;

% Read the binary data from the chip
data = read(S, numData);
```

7. Disconnect the SPI device and clean up by clearing the object.

```
disconnect(S);
clear('S');
```

Transmit Data Over SPI Using NI-845x

This example shows how to communicate with an EEPROM chip on a circuit board, with a board index of 0 and using port 0.

1 Ensure that the NI-845x adaptor is installed so that you can use the `spi` interface.

```
>> instrhwinfo('spi')

ans =

  HardwareInfo with properties:

    SupportedVendors: {'aardvark'  'ni845x'}
    InstalledVendors: {'ni845x'}
```

2 Look at the NI-845x adaptor properties.

```
>> instrhwinfo('spi', 'ni845x')

ans =

  HardwareInfo with properties:

              VendorName: 'ni845x'
       VendorDescription: 'National Instruments'
       VendorLibraryName: 'Ni845x.dll'
       InstalledBoardIds: {[0]}
      BoardSerialNumbers: {'0183388B'}
      ObjectConstructors: {'spi('ni845x', 0, 0)'}
```

Make sure that you have the NI-845x software driver installed. For details, see "SPI Interface Usage Requirements and Guidelines" on page 10-16.

3 Create the SPI object called `s2`, using these properties:

```
% Vendor = ni845x
% BoardIndex = 0
% Port = 0

s2 = spi('ni845x', 0, 0);
```

You must provide these three parameters to create the object.

4 Look at the object properties.

```
>> disp(s2)
SPI Object :

    Adapter Settings
        BoardIndex:             0
        BoardSerial:            0183388B
        VendorName:             ni845x

    Communication Settings
        BitRate:                1000000 Hz
        ChipSelect:             0
        ClockPhase:             FirstEdge
        ClockPolarity:          IdleLow
        Port:                   0

    Communication State
        ConnectionStatus:       Disconnected

    Read/Write State
        TransferStatus:         Idle
```

When you create the `spi` object, default communication settings are used, as shown here. To change any of these settings, see "Using Properties on the SPI Object" on page 10-13 for more information and a list of the properties.

5. Connect to the chip.

   ```
   connect(s2);
   ```

6. Read and write to the chip.

   ```
   % Create a variable containing the data to write
   dataToWrite = [3 0 0 0];

   % Write the binary data to the chip
   write(s2, dataToWrite);

   % Create a variable that contains the number of values to read
   numData = 5;

   % Read the binary data from the chip
   data = read(s2, numData);

   ans =

        0     0     0     0     0
   ```

7. Disconnect the SPI device and clean up by clearing the object.

   ```
   disconnect(s2);
   clear('s2');
   ```

SPI Functions

You can use these functions with the `spi` object.

Note SPI is a full duplex communication protocol, and data must be written in order to read data. You can use the `read` function to write dummy data to the device. The `write` function flushes the data returned by the device. The `writeAndRead` function does the read and write together.

Function	Purpose

`instrhwinfo`	Check that the Aardvark and/or NI-845x adaptor is installed.
	
	`instrhwinfo('spi')`
	
	Look at the adaptor properties.
	
	`instrhwinfo('spi', 'Aardvark')`
	
	`instrhwinfo('spi', 'ni845x')`

`spiinfo`	Returns information about devices and displays the information on a per vendor basis. `info = spiinfo()`
`connect`	Connect the SPI object to the device. Use this syntax: `connect(spiObject);`

`read`	Synchronously read binary data from the device. To read data, first create a variable, such as `numData`, to specify the size of the data to read. In this case, create the variable to read 5 bytes. Then use the `read` function as shown here, where `spiObject` is the name of your object. This process is also shown in step 6 of the previous example. The precision of the data is UINT8.
	
	`numData = 5;`
	
	`read(spiObject, numData);`
	
	Or you can use this syntax:
	
	`A = read(spiObject, size)`

Function	Purpose

`write`	Synchronously write binary data to the device. To write data, first create a variable, such as `dataToWrite`. In this case, create the data `[3 0 0 0]`. Then use the `write` function as shown here, where `spiObject` is the name of your object. This process is also shown in step 6 of the previous example. The precision of the data written is UINT8.
	
	`dataToWrite = [3 0 0 0];`
	
	`write(spiObject, dataToWrite);`

`writeAndRead`	Synchronously do a simultaneous read and write of binary data with the device. In this case, the function synchronously writes the data specified by the variable `dataToWrite` to the device in binary format, then synchronously reads from the device and returns the data to the variable `data`, as shown here, where `spiObject` is the name of your object. The precision of the data written and read is UINT8.
	
	`dataToWrite = [3 0 0 0];`
	
	`data = writeAndRead(spiObject, dataToWrite)`

`disconnect`	Disconnect SPI object from the device. Use this syntax: `disconnect(spiObject);`

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Using Properties on the SPI Object

Use the `properties` function on the `spi` object to see the available properties. In the preceding example, the syntax would be:

```
properties(S)
```

The following shows the output of the properties from the preceding example, “Transmitting Data Over the SPI Interface” on page 10-7.

```
>> properties(S)

Properties for class instrument.interface.spi.aardvark.Spi:

    BitRate
    ClockPhase
    ClockPolarity
    ChipSelect
    Port
    BoardIndex
    VendorName
    BoardSerial
    ConnectionStatus
    TransferStatus
```

You can use these interface-specific properties with the `spi` object.

Property	Description
`BitRate`	SPI clock speed. Must be a positive, nonzero value specified in Hz. The default is `1000000` Hz for both the Aardvark and NI-845x adaptors. To change from the default: `S.BitRate = 400000`
`ClockPhase`	SPI clock phase. Can be specified as `'FirstEdge'` or `'SecondEdge'`. The default of `'FirstEdge'` is used if you do not specify a phase. `ClockPhase` indicates when the data is sampled. If set to `'FirstEdge'`, the first edge of the clock is used to sample the first data byte. The first edge may be the rising edge (if `ClockPolarity` is set to `'IdleLow'`), or the falling edge (if `ClockPolarity` is set to `'IdleHigh'`). If set to `'SecondEdge'`, the second edge of the clock is used to sample the first data byte. The second edge may be the falling edge (if `ClockPolarity` is set to `'IdleLow'`), or the rising edge (if `ClockPolarity` is set to `'IdleHigh'`). To change from the default: `S.ClockPhase = 'SecondEdge'`

Property	Description
ClockPolarity	SPI clock polarity. Can be specified as 'IdleLow' or 'IdleHigh'. The default of 'IdleLow' is used if you do not specify a phase. ClockPolarity indicates the level of the clock signal when idle. 'IdleLow' means the clock idle state is low, and 'IdleHigh' means the clock idle state is high. To change from the default: `S.Polarity = 'IdleHigh'`
ChipSelect	SPI chip select line. The Aardvark adaptor uses 0 as the chip select line since it has only one line, so that is the default and only valid value.
Port	Use to create spi object. Port number of your hardware, specified as the number 0. The Aardvark adaptor uses 0 as the port number when there is one adaptor board connected. If there are multiple boards connected, they could use ports 0 and 1. Specify port number as the third argument when you create the spi object: `S = spi('aardvark', 0, 0);`
BoardSerial	Unique identifier of the SPI communication device.
VendorName	Use to create spi object. Adaptor board vendor, must be set to 'aardvark', for use with Total Phase Aardvark adaptor or 'ni845x' for use with the NI-845x adaptor. Specify the vendor as the first argument when you create the spi object: `S = spi('aardvark', 0, 0);`
BoardIndex	Use to create spi object. Specifies the board index of the hardware. Usually set to 0. Specify board index as the second argument when you create the spi object: `S = spi('aardvark', 0, 0);`
ConnectionStatus	Returns the connection status of the SPI object. Possible values are Disconnected (default) and Connected.
TransferStatus	Returns the read/write operation status of the SPI object. Possible values: Idle (default) – The device is not transferring any data. Read – The device is reading data. Write – The device is writing data. ReadWrite – The device is reading and writing data.

The properties all have defaults, as indicated in the table. You do not need to set a property unless you want to change it to a different value from the default. Aside from the three properties required to construct the object – `VendorName`, `BoardIndex`, and `Port` – any other property is set using the .dot notation syntax:

```
<object_name>.<property_name> = <value>
```

Here are a few examples of using this syntax.

```
Change the BitRate from the default of 1000000 to 500000 kHz

S.BitRate = 500000

Change the ClockPhase from the default of 'FirstEdge' to 'SecondEdge'

S.ClockPhase = 'SecondEdge'
```

where `S` is the name of the object used in the examples.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

SPI Interface Usage Requirements and Guidelines

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface. The following sections describe requirements for each option.

Aardvark-specific Requirements

To use the SPI interface with the Aardvark adaptor, you must download the Hardware Support Package to obtain the necessary files. You must also download the USB device driver from the vendor.

If you do not have the Aardvark driver installed, see “Install the Total Phase Aardvark I2C/SPI Interface Support Package” on page 15-9.

Install the Aardvark Software API and Shared Library appropriate for your operating system.

The `aardvark.dll` file that comes with the Total Phase Aardvark adaptor board must be available in one of these locations for use on Windows platforms:

- Location where MATLAB was started from (bin folder)
- MATLAB current folder (PWD)
- Windows folder `C:\winnt` or `C:\windows`
- Folders listed in the path environment variable

For use on Linux platforms, the `aardvark.so` file that comes with the Total Phase Aardvark adaptor board must be in your MATLAB path.

NI-845x-specific Requirements

To use the SPI interface with the NI-845x adaptor, you must download the hardware support package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this support package.

If you do not have the NI-845x driver installed, see “Install the NI-845x I2C/SPI Interface Support Package” on page 15-8 to install it.

Controlling Devices Using MODBUS

- “MODBUS Interface Supported Features” on page 11-2
- “Create a MODBUS Connection” on page 11-3
- “Configure Properties for MODBUS Communication” on page 11-5
- “Read Data from a MODBUS Server” on page 11-8
- “Read Temperature from a Remote Temperature Sensor” on page 11-13
- “Write Data to a MODBUS Server” on page 11-14
- “Write and Read Multiple Holding Registers” on page 11-16
- “Modify the Contents of a Holding Register Using a Mask Write” on page 11-18
- “Use the Modbus Explorer App” on page 11-19
- “Configure a Connection in the Modbus Explorer” on page 11-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 11-26
- “Control a PLC Using the Modbus Explorer” on page 11-28
- “Generate a Script from Your Modbus Explorer Session” on page 11-33

MODBUS Interface Supported Features

In this section...

"MODBUS Capabilities" on page 11-2
"Supported Platforms for MODBUS" on page 11-2

MODBUS Capabilities

Instrument Control Toolbox supports the MODBUS interface over TCP/IP or Serial RTU. You can use it to communicate with MODBUS servers, such as controlling a PLC (Programmable Logic Controller), communicating with a temperature controller, controlling a stepper motor, sending data to a DSP, reading bulk memory from a PAC controller, or monitoring temperature and humidly on a MODBUS probe.

Using the MODBUS interface, you can do the following tasks, which correspond to the MODBUS function codes listed in the table.

Functionality	MODBUS Function Code
Read and write coils	1, 5, 15
Read discrete inputs	2
Read and write holding registers	3, 6, 16
Read input registers	4
Perform mask writes on holding registers	22
Perform write/read (in one operation) on holding registers	23

Supported Platforms for MODBUS

Instrument Control Toolbox supports the MODBUS interface over TCP/IP or Serial RTU. It is supported on the following platforms.

- Linux 64-bit
- macOS 64-bit
- Microsoft Windows 64-bit

Note The Instrument Control Toolbox MODBUS support works on the MATLAB command line only. It is not available in the Test & Measurement Tool.

Create a MODBUS Connection

Instrument Control Toolbox supports the MODBUS interface over TCP/IP or Serial RTU. You can use it to communicate with MODBUS servers, such as a PLC. The typical workflow is:

- Create a MODBUS connection to a server or hardware.
- Configure the connection if necessary.
- Perform read and write operations, such as communicating with a temperature controller.
- Clear and close the connection.

To communicate over the MODBUS interface, you first create a MODBUS object using the `modbus` function. Creating the object also makes the connection. The syntax is:

```
<objname> = modbus('Transport', 'DeviceAddress')
```

or

```
<objname> = modbus('Transport', 'Port')
```

You must set the transport type as either `'tcpip'` or `'serialrtu'` to designate the protocol you want to use. Then set the address and port, as shown in the next sections. You can also use name-value pairs in the object creation to set properties such as `Timeout` and `ByteOrder`.

When you create the MODBUS object, it connects to the server or hardware. If the transport is `'tcpip'`, then `DeviceAddress` must be specified. Port is optional and defaults to 502 (reserved port for MODBUS). If the transport is `'serialrtu'`, then `'Port'` must be specified.

Create Object Using TCP/IP Transport

When the transport is `'tcpip'`, you must specify `DeviceAddress`. This is the IP address or host name of the MODBUS server. `Port` is the remote port used by the MODBUS server. Port is optional and defaults to `502`, which is the reserved port for MODBUS.

This example creates the MODBUS object `m` using the device address shown and `port` of `308`.

```
m = modbus('tcpip', '192.168.2.1', 308)

m = 

   Modbus TCPIP with properties:

    DeviceAddress: '192.168.2.1'
             Port: 308
           Status: 'open'
       NumRetries: 1
          Timeout: 10 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

Create Object Using Serial RTU Transport

When the transport is `'serialrtu'`, you must specify `'Port'`. This is the Serial port the MODBUS server is connected to.

This example creates the MODBUS object `m` using the `port` of `'COM3'`.

```
m = modbus('serialrtu','COM3')

m = 

   Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 10 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

Create Object and Set a Property

You can create the object using a name-value pair to set the properties such as `Timeout`. The `Timeout` property specifies the maximum time in seconds to wait for a response from the MODBUS server, and the default is `10`. You can change the value either during object creation or after you create the object.

For the list and description of properties you can set for both transport types, see "Configure Properties for MODBUS Communication" on page 11-5.

This example creates a MODBUS object using Serial RTU, but increases the `Timeout` to `20` seconds.

```
m = modbus('serialrtu','COM3','Timeout',20)


m = 

   Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 20 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

The output reflects the `Timeout` property change.

Configure Properties for MODBUS Communication

The `modbus` object has the following properties.

Property	Transport Type	Description
`'DeviceAddress'`	TCP/IP only	IP address or host name of MODBUS server, for example, `'192.168.2.1'`. Required during object creation if transport is TCP/IP. `m = modbus('tcpip', '192.168.2.1')`
`Port`	TCP/IP only	Remote port used by MODBUS server. The default is 502. Optional during object creation if transport is TCP/IP. `m = modbus('tcpip', '192.168.2.1', 308)`
`'Port'`	Serial RTU only	Serial port MODBUS server is connected to, for example, `'COM1'`. Required during object creation if transport is Serial RTU. `m = modbus('serialrtu','COM3')`
`Timeout`	Both TCP/IP and Serial RTU	Maximum time in seconds to wait for a response from the MODBUS server, specified as a positive value of type `double`. The default is `10`. You can change the value either during object creation, or after you create the object. `m.Timeout = 30;`
`NumRetries`	Both TCP/IP and Serial RTU	Number of retries to perform if there is no reply from the server after a timeout. If using the Serial RTU transport, the message is resent. If using the TCP/IP transport, the connection is closed and reopened. `m.NumRetries = 5;`
`'ByteOrder'`	Both TCP/IP and Serial RTU	Byte order of values written to or read from 16-bit registers. Valid choices are `'big-endian'` and `'little-endian'`. The default is `'big-endian'`, as specified by the MODBUS standard. `m.ByteOrder = 'little-endian';`
`'WordOrder'`	Both TCP/IP and Serial RTU	Word order for register reads and writes that span multiple 16-bit registers. Valid choices are `'big-endian'` and `'little-endian'`. The default is `'big-endian'`, and it is device-dependent. `m.WordOrder = 'little-endian';`

Property	Transport Type	Description
`BaudRate`	Serial RTU only	Bit transmission rate for serial port communication. Default is 9600 bits per seconds, but the actual required value is device-dependent. `m.Baudrate = 28800;`
`DataBits`	Serial RTU only	Number of data bits to transmit. Default is 8, which is the MODBUS standard for Serial RTU. Other valid values are 5, 6, and 7. `m.DataBits = 6;`
`Parity`	Serial RTU only	Type of parity checking. Valid choices are `'none'` (default), `'even'`, `'odd'`, `'mark'`, and `'space'`. The actual required value is device-dependent. If set to the default of `none`, parity checking is not performed, and the parity bit is not transmitted. `m.Parity = 'odd';`
`StopBits`	Serial RTU only	Number of bits used to indicate the end of data transmission. Valid choices are 1 (default) and 2. Actual required value is device-dependent, though 1 is typical for even/odd parity and 2 for no parity. `m.StopBits = 2;`

Set a Property During Object Creation

You can change property values either during object creation or after you create the object.

You can create the `modbus` object using a name-value pair to set a value during object creation.

This example creates the MODBUS object and increases the `Timeout` to `20` seconds.

```
m = modbus('serialrtu','COM3','Timeout',20)


m = 

   Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 20 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

The output reflects the `Timeout` property change from the default of `10` seconds to `20` seconds.

Set a Property After Object Creation

You can change a property anytime by setting the property value using this syntax after you have created a MODBUS object.

```
<object_name>.<property_name> = <property_value>
```

This example using the same object named `m` increases the `Timeout` to 30 seconds.

```
m = modbus('serialrtu','COM3');
m.Timeout = 30
```

This example changes the `Parity` from the default of `none` to `even`.

```
m = modbus('serialrtu','COM3');
m.Parity = 'even';
```

Read Data from a MODBUS Server

In this section...

"Types of Data You Can Read Over MODBUS" on page 11-8
"Reading Coils Over MODBUS" on page 11-8
"Reading Inputs Over MODBUS" on page 11-9
"Reading Input Registers Over MODBUS" on page 11-9
"Reading Holding Registers Over MODBUS" on page 11-10
"Specifying Server ID and Precision" on page 11-10
"Reading Mixed Data Types" on page 11-11

Types of Data You Can Read Over MODBUS

The `read` function performs read operations from four types of target-addressable areas:

- Coils
- Inputs
- Input registers
- Holding registers

When you perform the read, you must specify the target type (`target`), the starting address (`address`), and the number of values to read (`count`). You can also optionally specify the address of the server (`serverId`) for any target type, and the data format (`precision`) for registers.

For an example showing the entire workflow of reading a holding register on a PLC, see "Read Temperature from a Remote Temperature Sensor" on page 11-13.

Reading Coils Over MODBUS

If the read target is coils, the function reads the values from 1–2000 contiguous coils in the remote server, starting at the specified address. A coil is a single output bit. A value of `1` indicates the coil is on and a value of `0` means it is off.

The syntax to read coils is:

```
read(obj,'coils',address,count)
```

The `obj` parameter is the name of the MODBUS object. The examples assume you have created a MODBUS object, `m`. For information on creating the object, see "Create a MODBUS Connection" on page 11-3.

The `address` parameter is the starting address of the coils to read, and it is a double. The `count` parameter is the number of coils to read, and it is a double. If the read is successful, it returns a vector of doubles, each with the value `1` or `0`, where the first value in the vector corresponds to the coil value at the starting address.

This example reads 8 coils, starting at address 1.

```
read(m,'coils',1,8)
```

```
ans =

   1   1   0   1   1   0   1   0
```

You can also create a variable to be read later.

```
data = read(m,'coils',1,8)

data =

   1   1   0   1   1   0   1   0
```

Reading Inputs Over MODBUS

If the read target is inputs, the function reads the values from 1–2000 contiguous discrete inputs in the remote server, starting at the specified address. A discrete input is a single input bit. A value of `1` indicates the input is on, and a value of `0` means it is off.

The syntax to read inputs is:

```
read(obj,'inputs',address,count)
```

The `obj` parameter is the name of the MODBUS object. The examples assume you have created a MODBUS object, `m`. For information on creating the object, see “Create a MODBUS Connection” on page 11-3.

The `address` parameter is the starting address of the inputs to read, and it is a double. The `count` parameter is the number of inputs to read, and it is a double. If the read operation is successful, it returns a vector of doubles, each with the `1` or `0`, where the first value in the vector corresponds to the input value at the starting address.

This example reads 10 discrete inputs, starting at address 2.

```
read(m,'inputs',2,10)

ans =

   1   1   0   1   1   0   1   0   0   1
```

Reading Input Registers Over MODBUS

If the read target is input registers, the function reads the values from 1–125 contiguous input registers in the remote server, starting at the specified address. An input register is a 16-bit read-only register.

The syntax to read input registers is:

```
read(obj,'inputregs',address,count)
```

The `obj` parameter is the name of the MODBUS object. The examples assume you have created a MODBUS object, `m`. For information on creating the object, see “Create a MODBUS Connection” on page 11-3.

The `address` parameter is the starting address of the input registers to read, and it is a double. The `count` parameter is the number of input registers to read, and it is a double. If the read operation is

successful, it returns a vector of doubles. Each double represents a 16-bit register value, where the first value in the vector corresponds to the input register value at the starting address.

This example reads 4 input registers, starting at address 20.

```
read(m,'inputregs',20,4)

ans =

   27640   60013   51918   62881
```

Reading Holding Registers Over MODBUS

If the read target is holding registers, the function reads the values from 1–125 contiguous holding registers in the remote server, starting at the specified address. A holding register is a 16-bit read/write register.

The syntax to read inputs is:

```
read(obj,'holdingregs',address,count)
```

The `obj` parameter is the name of the MODBUS object. The examples assume you have created a MODBUS object, `m`. For information on creating the object, see “Create a MODBUS Connection” on page 11-3.

The `address` parameter is the starting address of the holding registers to read, and it is a double. The `count` parameter is the number of holding registers to read, and it is a double. If the read operation is successful, it returns a vector of doubles. Each double represents a 16-bit register value, where the first value in the vector corresponds to the holding register value at the starting address.

This example reads 4 holding registers, starting at address 20.

```
read(m,'holdingregs',20,4)

ans =

   27640   60013   51918   62881
```

For an example showing the entire workflow of reading a holding register on a PLC, see “Read Temperature from a Remote Temperature Sensor” on page 11-13.

Specifying Server ID and Precision

You can read any of the four types of targets and also specify the optional parameters for server ID, and can specify precision for registers.

Server ID Option

The `serverId` argument specifies the address of the server to send the read command to. Valid values are `0–247`, with `0` being the broadcast address. This argument is optional, and the default is `1`.

Note What some devices refer to as a `slaveID` property, may work as a `serverID` property in the MODBUS interface. For some manufacturers a slave ID is sometimes referred to as a server ID. If

your device uses a `slaveID` property, it might work to use it as the `serverID` property with the `read` command as described here.

The syntax to specify server ID is:

```
read(obj,target,address,count,serverId)
```

This example reads 8 coils starting at address 1 from server ID 3.

```
read(m,'coils',1,8,3);
```

Precision Option

The `'precision'` argument specifies the data format of the register being read from on the MODBUS server. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

Note that `'precision'` does not refer to the return type, which is always `'double'`. It only specifies how to interpret the register data.

The syntax to specify precision is:

```
read(obj,target,address,count,precision)
```

This example reads 8 holding registers starting at address 1 using a precision of `'uint32'`.

```
read(m,'holdingregs',1,8,'uint32');
```

Both Options

You can set both the `serverId` option and the `'precision'` option together when the target is a register. When you use both options, the `serverId` should be listed first after the required arguments.

The syntax to specify both Server ID and precision is:

```
read(obj,target,address,count,serverId,precision)
```

This example reads 8 holding registers starting at address 1 using a precision of `'uint32'` from Server ID 3.

```
read(m,'holdingregs',1,8,3,'uint32');
```

Reading Mixed Data Types

You can read contiguous values of different data types (precisions) in a single read operation by specifying the data type for each value. You can do that within the syntax of the `read` function, or set up variables containing arrays of counts and precisions. Both methods are shown below.

Within the `read` Syntax

The syntax of the `read` function is as follows:

```
read(m,'holdingregs',500,10,'uint32');
```

In that example, the target type is holding registers, the starting address is `500`, the count is `10`, and the precision is `uint32`. If you wanted to have the 10 values be of mixed data types, you can use this syntax:

```
read(m,'holdingregs',500,[3 2 3 2],{'uint16', 'single', 'double', 'int16'});
```

You use an array of values within the command for both count and precision. In this case, the counts are 3, 2, 3, and 2. The command will read 3 values of data type `uint16`, 2 values of data type `single`, 3 values of data type `double`, and 2 values of data type `int16`. The registers are contiguous, starting at address `500`. In this example it reads 3 `uint16` values from address `500-502`, 2 `single` values from address `503-506`, 3 `double` values from address `507-518`, and 2 `int16` values from address `519-520`, all in one read operation.

Using Variables

Instead of using arrays inside the read command as shown above, you can also use arrays as variables in the command. The equivalent for the example shown above is:

```
count = [3 2 3 2]
precision = {'uint16', 'single', 'double', 'int16'}
read(m,'holdingregs',500,count,precision);
```

Using variables is convenient when you have a lot of values to read and they are of mixed data types.

Read Temperature from a Remote Temperature Sensor

This example shows how to read temperature and humidity measurements from a remote sensor on a PLC connected via TCP/IP. The temperature sensor is connected to a holding register at address 1 on the board, and the humidity sensor is at address 5.

1. Create the MODBUS object, using TCP/IP.

```
m = modbus('tcpip', '192.168.2.1', 502)

m = 

   Modbus TCPIP with properties:

    DeviceAddress: '192.168.2.1'
             Port: 502
           Status: 'open'
       NumRetries: 1
          Timeout: 10 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

2. The humidity sensor does not always respond instantly, so increase the timeout value to 20 seconds.

```
m.Timeout = 20
```

3. The temperature sensor is connected to a holding register at address 1 on the board. Read 1 value to get the current temperature reading. Since temperature value is a double, set the precision to a double.

```
read(m,'holdingregs',1,1,'double')

ans =

   46.7
```

4. The humidity sensor is connected to the holding register at address 5 on the board. Read 1 value to get the current humidity reading.

```
read(m,'holdingregs',5,1,'double')

ans =

   35.8
```

5. Disconnect from the server and clear the object.

```
clear m
```

Write Data to a MODBUS Server

In this section...
"Types of Data You Can Write to Over MODBUS" on page 11-14
"Writing Coils Over MODBUS" on page 11-14
"Writing Holding Registers Over MODBUS" on page 11-14

Types of Data You Can Write to Over MODBUS

The `write` function performs write operations to two types of target addressable areas:

- Coils
- Holding registers

Each of the two areas can accept a write request to a single address or a contiguous address range. When you perform the write operation, you must specify the target type (`target`), the starting address (`address`), and the values to write (`values`). You can also optionally specify the address of the server (`serverId`) and the data format (`precision`).

Writing Coils Over MODBUS

If the write target is coils, the function writes a contiguous sequence of 1–1968 coils to either on or off (1 or 0) in a remote device. A coil is a single output bit. A value of `1` indicates the coil is on, and a value of `0` means it is off.

The syntax to write to coils is:

```
write(obj,'coils',address,values)
```

The `obj` parameter is the name of the MODBUS object. The examples assume you have created a MODBUS object, `m`. For information on creating the object, see "Create a MODBUS Connection" on page 11-3.

The `address` parameter is the starting address of the coils to write to, and it is a double. The `values` parameter is an array of values to write. For a target of coils, valid values are `0` and `1`.

This example writes to 4 coils, starting at address 8289.

```
write(m,'coils',8289,[1 1 0 1])
```

You can also create a variable for the values to write.

```
values = [1 1 0 1];
write(m,'coils',8289,values)
```

Writing Holding Registers Over MODBUS

If the write target is holding registers, the function writes a block of 1–123 contiguous registers in a remote device. Values whose representation is greater than 16 bits are stored in consecutive register addresses.

The syntax to write to holding registers is:

```
write(obj,'holdingregs',address,values)
```

The `obj` parameter is the name of the MODBUS object. The examples assume you have created a MODBUS object, `m`. For information on creating the object, see “Create a MODBUS Connection” on page 11-3.

The `address` parameter is the starting address of the holding registers to write to, and it is a double. The `values` parameter is an array of values to write. For a target of holding registers, valid values must be in the range of the specified precision.

This example sets the register at address 49153 to 2000.

```
write(m,'holdingregs',49153,2000)
```

Precision Option

The `'precision'` argument specifies the data format of the register being written to on the MODBUS server. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

The values passed in to be written are converted to register values based on the specified precision. For precision values `'int32'`, `'uint32'`, and `'single'`, each value corresponds to 2 registers, and for `'uint64'`, `'int64'` and `'double'`, each value corresponds to 4 registers. For `'int16'` and `'uint16'`, each value is from a single 16-bit register.

This example writes 3 values, starting at address 29473 and converting to `single` precision.

```
write(m,'holdingregs',29473,[928.1 50.3 24.4],'single')
```

Write and Read Multiple Holding Registers

The writeRead function is used to perform a combination of one write operation and one read operation on groups of holding registers in a single MODBUS transaction. The write operation is always performed before the read. The range of addresses to read and the range of addresses to write must be contiguous, but each is specified independently and may or may not overlap.

The syntax for the write-read operation to holding registers is:

```
writeRead(obj,writeAddress,values,readAddress,readCount)
```

The obj parameter is the name of the MODBUS object. The examples assume you have created a MODBUS object, m. For information on creating the object, see “Create a MODBUS Connection” on page 11-3.

The writeAddress is the starting address of the holding registers to write to, and it is a double. The values parameter is an array of values to write. The first value in the vector is written to the writeAddress. Each value must be in the range 0–65535.

The readAddress is the starting address of the holding registers to read, and readCount is the number of registers to read.

If the operation is successful, it returns an array of doubles, each representing a 16-bit register value, where the first value in the vector corresponds to the register value at address specified in readAddress.

This example writes 2 holding registers starting at address 601, and reads 4 holding registers starting at address 19250.

```
writeRead(m,601,[1024 512],19250,4)

ans =

   27640   60013   51918   62881
```

You can optionally create variables for the values to be written, instead of including the array of values in the function syntax, as shown above. The same example could be written this way, using a variable for the values:

```
values = [1024 512];
writeRead(m,601,values,19250,4)

ans =

   27640   60013   51918   62881
```

Server ID Option

The serverId argument specifies the address of the server to send the read command to. Valid values are 0–247, with 0 being the broadcast address. This argument is optional, and the default is 1.

Note What some devices refer to as a slaveID property, may work as a serverID property in the MODBUS interface. For some manufacturers a slave ID is sometimes referred to as a server ID. If your device uses a slaveID property, it might work to use it as the serverID property with the writeRead command as described here.

The syntax to specify server ID is:

```
writeRead(obj,writeAddress,values,readAddress,readCount,serverId)
```

This example writes 3 holding registers starting at address 400, and reads 4 holding registers starting at address 52008, from server ID 6.

```
writeRead(m,400,[1024 512 680],52008,4,6)

ans =

   38629   84735   29456   39470
```

Precision Option

The `'writePrecision'` and `'readPrecision'` arguments specify the data format of the register being read from or written to on the MODBUS server. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

The values passed in to be written are converted to register values based on the specified precision. For precision values `'int32'`, `'uint32'`, and `'single'`, each value corresponds to 2 registers, and for `'uint64'`, `'int64'` and `'double'`, each value corresponds to 4 registers. For `'int16'` and `'uint16'`, each value is from a single 16-bit register.

Note that precision specifies how to interpret or convert the register data, not the return type of the read operation. The data returned is always of type double.

The syntax for designating the write or read precision is:

```
writeRead(obj,writeAddress,values,writePrecision,readAddress,readCount,readPrecision)
```

If you want to use the `serverId` argument as well, it goes after the `readPrecision`.

This example writes 3 holding registers starting at address 400 and reads 4 holding registers starting at address 52008, from server ID 6. It also specifies a `writePrecision` of `'uint64'` and a `readPrecision` of `'uint32'`.

```
writeRead(m,400,[1024 512 680],'uint64',52008,4,'uint32',6)

ans =

   38629   84735   29456   39470
```

This example reads 2 holding registers starting at address 919, and writes 3 holding registers starting at address 719, formatting read and write for single precision data registers.

```
  values = [1.14 5.9 11.27];
  writeRead(m,719,values,'single',919,2,'single')
```

Modify the Contents of a Holding Register Using a Mask Write

You can modify the contents of a holding register using the `maskWrite` function. The function can set or clear individual bits in a specific holding register. It is a read/modify/write operation, and uses a combination of an AND mask, an OR mask, and the current contents of the register.

The function algorithm works as follows:

```
Result = (register value AND andMask) OR (orMask AND (NOT andMask))
```

For example:

```
                 Hex    Binary
Current contents  12   0001 0010
And_Mask          F2   1111 0010
Or_Mask           25   0010 0101
(NOT And_Mask)    0D   0000 1101

Result            17   0001 0111
```

If the `orMask` value is 0, the result is simply the logical ANDing of the current contents and the `andMask`. If the `andMask` value is 0, the result is equal to the `orMask` value.

The contents of the register can be read by using the `read` function with the target set to `'holdingregs'`. They could, however, be changed subsequently as the controller scans its user logic program.

The syntax for the mask write operation for holding registers is:

```
maskWrite(obj, address, andMask, orMask)
```

If you want to designate a server ID, use:

```
maskWrite(obj, address, andMask, orMask, serverId)
```

The `obj` parameter is the name of the MODBUS object. The examples assume you have created a MODBUS object, `m`. For information on creating the object, see "Create a MODBUS Connection" on page 11-3.

The `address` is the register address to perform mask write on. The `andMask` parameter is the `AND` value to use in the mask write operation. The valid range is `0–65535`. The `orMask` parameter is the `OR` value to use in the mask write operation. The valid range is `0–65535`.

This example sets bit 0 at address 20 and performs a mask write operation. Since the `andMask` is a 6, that clears all bits except for bits 1 and 2. Bits 1 and 2 are preserved.

```
andMask = 6
orMask = 0

maskWrite(m,20,andMask,orMask)
```

Use the Modbus Explorer App

You can read and write to coils and registers in the Modbus Explorer app. The app supports a subset of the MATLAB Modbus functionality. You can do the following in the Modbus Explorer:

- Read coils, inputs, registers, and holding registers. This is the functionality of the Modbus `read` function.
- Write to coils and holding registers. This is the functionality of the Modbus `write` function.

The app does not support the functionality of the Modbus `writeRead` function or the `maskWrite` function.

The Modbus Explorer offers a user interface to easily set up reads and writes and a live plot to see the values. The read table allows you to easily organize and manage reads for multiple addresses, such as different sensors on a PLC.

To launch the Modbus Explorer, do one of these:

- In the MATLAB Apps tab, under **Test & Measurement**, select **Modbus Explorer**.
- At the MATLAB command line, type `modbusExplorer`.

To use the Modbus Explorer, you need to configure your device and connect over TCP/IP or Serial RTU. For information about how to configure and connect to your device, see “Configure a Connection in the Modbus Explorer” on page 11-20.

After you have successfully configured your device in the **Configure** tab, press the **Confirm Parameters** button to open the read and write section of the Modbus Explorer. You can then perform reads and writes. For information about doing reads, see “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23. For information about doing writes, see “Write to Coils and Holding Registers in the Modbus Explorer” on page 11-26.

See Also

Related Examples

- “Configure a Connection in the Modbus Explorer” on page 11-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 11-26
- “Control a PLC Using the Modbus Explorer” on page 11-28
- “Generate a Script from Your Modbus Explorer Session” on page 11-33

Configure a Connection in the Modbus Explorer

The first step in using the Modbus Explorer to communicate with a PLC or other Modbus device is to configure the communication with the device, either over TCP/IP or Serial RTU.

Communicate Over TCP/IP

1. Open the Modbus Explorer. On the MATLAB Apps tab, under **Test & Measurement**, select **Modbus Explorer**.
2. Choose your communication interface in the Modbus Explorer by clicking **Device** then **Modbus TCP/IP**.
3. On the **Configure** tab, configure the connection to your device by setting the following TCP/IP communication parameters in the toolstrip:

   Device Address: IP address of Modbus server, for example 192.168.2.20. This parameter is required to make the connection.

   Port: Remote port used by Modbus server. The default is 502. Change it if using a different port number.

   Timeout: Maximum time in seconds to wait for a response from the Modbus server, specified as a positive value. The default is 3. You can edit the value to increase or decrease the timeout. Note that the default when using the `Timeout` property programmatically is 10 seconds. If your device requires more than the default of 3 seconds in the app, increase the value.

   Byte Order: Byte order of values written to or read from 16-bit registers. The default is Big Endian, as specified by the Modbus standard. If your device requires Little Endian, change the value in the drop-down.

   Word Order: Word order for register reads and writes that span multiple 16-bit registers. The default is Big Endian, and it is device-specific. If your device requires Little Endian, change the value in the drop-down.
4. Configure the reading of data from your device by setting the following read parameters in the toolstrip:

   Server ID: Address of the server to send the read command to. If you do not specify a Server ID, the default of 1 is used. Valid values are 1-247.

   Register Type: Target type to read. You can perform a Modbus read operation on four types of targets: coils, inputs, input registers, and holding registers.

   Register Address: Starting address to read from, specified as a double. Enter the number for your starting address.

   Precision: Data format of the register being read from on the Modbus server. For coils and inputs, the precision is always `bit`. For holding registers and input resisters, you can specify precisions such as `uint16`.
5. To test the configuration, click **Read**. If your configuration parameters are correct, the read is successful and the **Read Value** populates with the value from the read. If you see `'ERROR'` in the **Read Value** field, adjust the parameters until the read is successful.

   This value needs to match the value listed in your device manual. Make sure this value and the other configuration parameters match the specifications for your device.
6. Once you have a correct read value, click **Confirm Parameters**. The rest of the tab appears, and your device is listed in the **Device List** on the left side of the app.
7. The register details you enter in the **Configure** tab are shown in the first row of the register table. You then use the table to set up reads from your device, or press **Import** to import a table of information that you previously exported.

For information about setting up reads, see "Read Coils, Inputs, and Registers in the Modbus Explorer" on page 11-23.

Communicate Over Serial RTU

1. Open the Modbus Explorer. In the MATLAB Apps tab, under **Test & Measurement**select, **Modbus Explorer**.
2. Choose your communication interface in the Modbus Explorer by clicking **Device** then **Modbus Serial**.
3. On **Configure** tab, configure the connection to your device by setting the following Serial RTU communication parameters in the toolstrip:

   Port: Serial port Modbus server is connected to, for example COM1.

   Baud Rate: Bit transmission rate for serial port communication. The default is 9600 bits per seconds, but the actual required value is device-dependent. Change the value in the drop-down if your device requires a different baud rate. Enter your baud rate value if it is not in the list.

   Parity: Type of parity checking. Valid choices are none (default), even, and odd. The actual required value is device-dependent. If set to the default of none, parity checking is not performed, and the parity bit is not transmitted.

   Stop Bits: Number of bits used to indicate the end of data transmission. Valid choices are 1 (default) and 2. The required value is device-dependent, though 1 is typical for even/odd parity and 2 for no parity.

   Data Bits: Number of data bits to transmit. The default is 8, which is the Modbus standard for Serial RTU. Other valid values are 5, 6, and 7.

   Timeout: Maximum time in seconds to wait for a response from the Modbus server, specified as a positive value. The default is 3. You can edit the value to increase or decrease the timeout. Note that the default when using the `Timeout` property programmatically is 10 seconds. If your device requires more than the default of 3 seconds in the app, increase the value.

   Byte Order: Byte order of values written to or read from 16-bit registers. The default is Big Endian, as specified by the Modbus standard. If your device requires Little Endian, change the value in the drop-down.

   Word Order: Word order for register reads and writes that span multiple 16-bit registers. The default is Big Endian, and it is device-specific. If your device requires Little Endian, change the value in the drop-down.
4. Configure the reading of data from your device by setting the following read parameters in the toolstrip:

   Server ID: Address of the server to send the read command to. If you do not specify a Server ID, the default of 1 is used. Valid values are 1-247.

   Register Type: Target type to read. You can perform a Modbus read operation on four types of targets: coils, inputs, input registers, and holding registers. Use the drop-down to select your type.

   Register Address: Starting address to read from, specified as a double. Enter the number for your starting address.

   Precision: Data format of the register being read from on the Modbus server. For coils and inputs, the precision is always `bit`. For holding registers and input resisters, you can specify precisions such as `uint16`.
5. To test the configuration, click **Read**. If your configuration parameters are correct, the read is successful and the **Read Value** fills in with the value from the read. If you see `'ERROR'` in the **Read Value** field, adjust the parameters until the read is successful.

This value needs to match the value listed in your device manual. Make sure this value and the other configuration parameters match the specifications for your device.

6 Once you have a correct read value, click **Confirm Parameters**. The rest of the tab appears, and your device is listed in the **Device List** on the left side of the app.

7 The register details you enter in the **Configure** tab are shown in the first row of the register table. You then use the table to set up reads from your device, or press **Import** to import a table of information that you previously exported.

For information about setting up reads, see “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23.

See Also

Related Examples

- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 11-26
- “Control a PLC Using the Modbus Explorer” on page 11-28
- “Generate a Script from Your Modbus Explorer Session” on page 11-33

Read Coils, Inputs, and Registers in the Modbus Explorer

You can read coils, inputs, input registers, and holding registers in the Modbus Explorer. This is the functionality of the Modbus `read` function.

You must first configure your device before performing read operations. For information on how to configure and connect to your device, see “Configure a Connection in the Modbus Explorer” on page 11-20.

1. To perform a read operation, enter the information about the coils, inputs, registers, or holding registers you want to control in the **Read Registers** table. The first row of the table is already filled in with the register you configured on the **Configure** tab.
2. To set up another register, click **Insert**. For each inserted row, click the **Address** field and enter the address of the coil, input, input register, or holding register you want to read values from.
3. In the **Register Type** column, click the down arrow to select the target type to read. You can perform a Modbus read operation on four types of targets: coils, inputs, input registers, and holding registers.
4. In the **Precision** column, click the down arrow to select the precision. Choose the data format of the register being read from on the Modbus server.
5. In the **Name** column, enter a name. A default name is provided, and you can keep it or change it.
6. Once you have entered all of the fields, click **Resume Reads** to start reading.

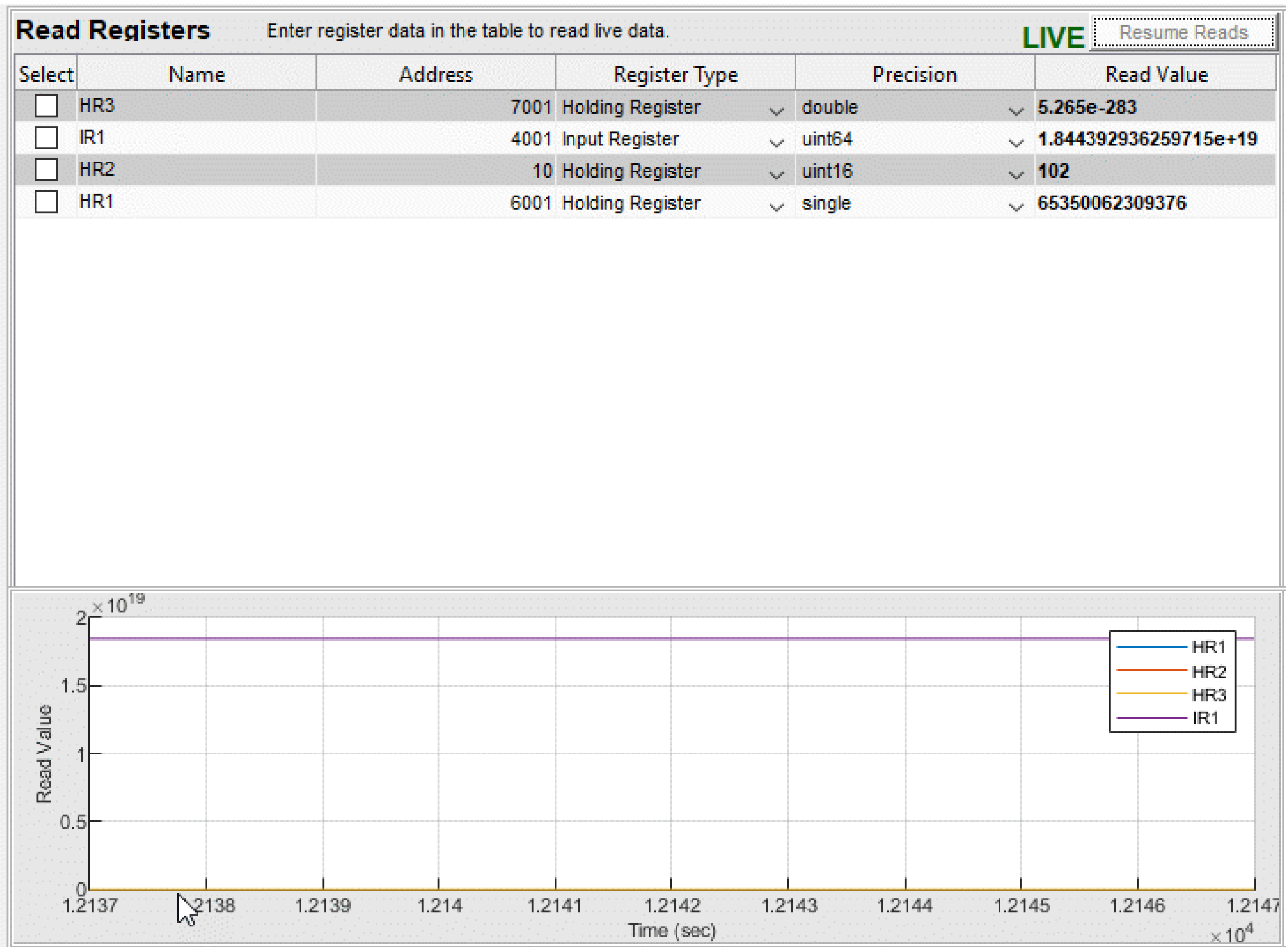
   If the read is successful, the **Read Value** column fills in with the read value. If you see `'ERROR'` in the **Read Value** field, adjust the parameters until the read is successful.
7. To read multiple registers, add rows to the table by clicking **Insert** on the toolstrip. New rows are added to the top of the table. You can add as many rows as you need. For each row, fill in all of the fields for that read. While you are editing a row, reading is paused, as indicated by the `PAUSED` status to the left of the **Resume Reads** button.

Read Registers Click 'Resume Reads' to start reading live data. PAUSED Resume Reads

Select	Name	Address	Register Type	Precision	Read Value
☐	HR3	7001	Holding Register	double	
☐	IR1	4001	Input Register	uint64	
☐	HR2	10	Holding Register	uint16	
☐	HR1	6001	Holding Register	single	

8. After you set up the table, click **Resume Reads**.

   The reads are performed and the status changes to `LIVE`. You can also see the read values in the live plot.



9 If you want to alter the plot, use the **Plot Tools** section. You can change the axes, show or hide the legend, and select which registers to display in the plot. The plot changes dynamically when you change any of these factors using the plot tools.

Edit the Read Registers Table

You can manipulate the register table by using the buttons in the toolstrip and the check boxes in the table.

- **Insert** - Insert a blank row at the top of the table.
- **Delete** - Delete selected rows. Use the check boxes in the table to select rows to delete.
- **Move Up** - Move up selected rows one position in the table. Use the check boxes in the table to select rows to move.
- **Move Down** - Move down selected rows one position in the table. Use the check boxes in the table to select rows to move.
- **Sort** - Sort the register table data by column value. Click **Sort** and choose **Name** to sort in alphabetic order of name, **Address** for ascending order of address, **Register Type** for alphabetical order of register type, or **Precision** for alphabetical order of precision.

Import or Export Read Data

You can export the contents of the **Read Registers** table to use again later.

To export the register table configuration, click the **Export** button in the toolstrip. In the Save dialog box, choose a file name and navigate to the save location, then click **Save**. The table is saved as a MAT-file.

To import the contents back into the **Read Registers** table, click **Import** in the toolstrip. In the Load dialog box, navigate to the saved MAT-file, select it, and click **Open**.

See Also

Related Examples

- “Configure a Connection in the Modbus Explorer” on page 11-20
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 11-26
- “Control a PLC Using the Modbus Explorer” on page 11-28
- “Generate a Script from Your Modbus Explorer Session” on page 11-33

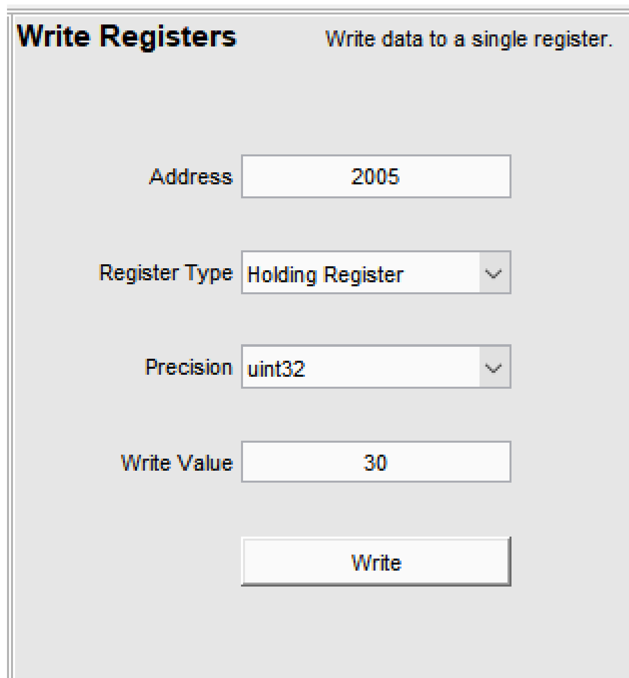
Write to Coils and Holding Registers in the Modbus Explorer

You can write to coils and holding registers in the Modbus Explorer. This is the functionality of the Modbus `write` function.

You must first configure your device before performing write or read operations. For information on how to configure and connect to your device, see “Configure a Connection in the Modbus Explorer” on page 11-20. For more information on performing reads, see “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23.

Use the **Write Registers** section of the app to perform write operations.

1. Enter the information about the register you want to control in the **Write Registers** section. You can do one write at a time using this section of the app.
2. To set up the write, click the **Address** field and enter the address of the register you want to write a value to.
3. In the **Register Type** field, click the down arrow to select the register type of the address. You can perform a Modbus write operation on two types of targets: coils and holding registers.
4. In the **Precision** field, click the down arrow to select the precision. This is the data format of the register being written to on the Modbus server.
5. In the **Write Value** field, enter the value to write. For coils and inputs, you can only write values of `0` or `1`. For input registers and holding registers you can write other values.
6. Once you have entered all the fields, the **Write** button becomes activated.



7. To send the value to the register, click **Write**.
8. If you have the same register listed in the **Read Registers** table, you see the read value update when you click **Write**. In the example shown here, you can see that the value of 30 was sent to the register and that it is now reflected in the read table for `Reg_5` in the first row.

Read Registers Enter register data in the table to read live data. LIVE Resume Reads

Select	Name	Address	Register Type	Precision	Read Value
☐	Reg_5	2005	Holding Register	uint32	30
☐	Reg_4	7001	Holding Register	double	5.265e-283
☐	Reg_3	4001	Input Register	uint64	1.844392936259715e+19
☐	Reg_2	10	Holding Register	uint16	102
☐	Reg_1	6001	Holding Register	single	65350062309376

Write Registers Write data to a single register.

Address 2005

Register Type Holding Register

Precision uint32

Write Value 30

Write

See Also

Related Examples

- “Configure a Connection in the Modbus Explorer” on page 11-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23
- “Control a PLC Using the Modbus Explorer” on page 11-28
- “Generate a Script from Your Modbus Explorer Session” on page 11-33

Control a PLC Using the Modbus Explorer

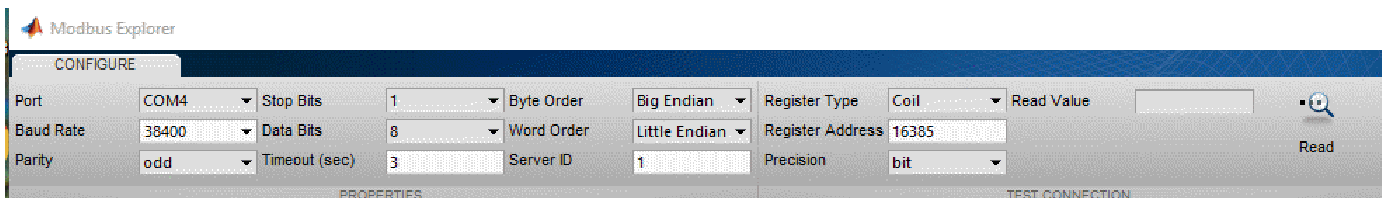
This example shows how to perform read and write operations to a PLC using the Modbus Explorer. The PLC is a Click Koyo cube with registers that can be used in industrial control and other industrial applications including controlling switches, timers, and sensors.

1. Open the Modbus Explorer. In the MATLAB Apps tab, under **Test & Measurement**, select **Modbus Explorer**.
2. The device is accessed over Serial RTU. To choose the communication interface in the Modbus Explorer click **Device** then **Modbus Serial** in the toolstrip.
3. On the **Configure** tab, configure the connection to your device by setting the following Serial RTU communication parameters in the toolstrip:

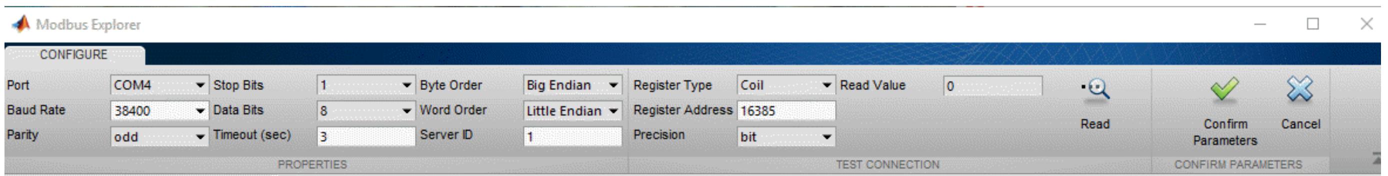
   Port: Serial port Modbus server is connected to. Set to `COM4`.
   Baud Rate: Bit transmission rate for serial port communication. The default is 9600 bits per seconds. Change it to 38400.
   Parity: Type of parity checking. Valid choices are none (default), even, and odd, and the actual required value is device-dependent. Set it to odd.
   Stop Bits: Number of bits used to indicate the end of data transmission. Valid choices are 1 (default) and 2, and the actual required value is device-dependent. Keep the default.
   Data Bits: Number of data bits to transmit. The default is 8, which is the Modbus standard for Serial RTU. Other valid values are 5, 6, and 7. Keep the default.
   Timeout: Maximum time in seconds to wait for a response from the Modbus server. The default is 3. You can edit the value to increase or decrease the timeout. Keep the default.
   Byte Order: Byte order of values written to or read from 16-bit registers. The default is Big Endian, as specified by the Modbus standard. Keep the default.
   Word Order: Word order for register reads and writes that span multiple 16-bit registers. The default is Big Endian, and it is device-specific. Set it to Little Endian.
4. Configure the reading of data from your device by setting the following read parameters in the toolstrip:

   Server ID: Address of the server to send the read command to, specified as a double. Valid values are 0-247, with 0 being the broadcast address. Set to 1.
   Register Type: Target area to read. You can perform a Modbus read operation on four types of targets: coils, inputs, input registers, and holding registers. Use the drop-down to select Coil.
   Register Address: Starting address to read from, specified as a double. Enter the number for your starting address, `16385` in this case.
   Precision: Data format of the register being read from on the Modbus server. For coils and inputs, the precision is always `bit`. For holding registers and input resisters, you can specify precisions such as `uint16`.

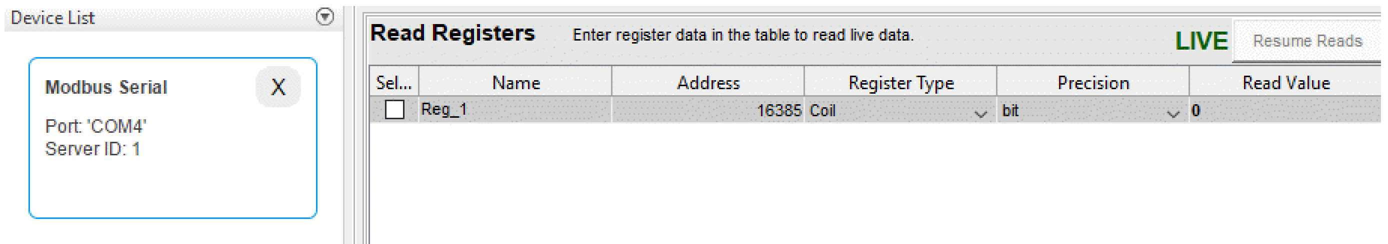
   The configuration should look like this after you configure the communication and read settings.



5. To test the configuration, click **Read**. If your configuration parameters are correct, the read is successful and the **Read Value** populates with the value from the read operation. If you get an error, adjust the parameters until the read is successful. In this case, the value should be 0.



6 After you have a correct read value, click **Confirm Parameters**. The **Configure** tab disappears and the **Modbus Explorer** tab appears, and your device is listed in the **Device List** on the left side of the app, as shown here.



7 You then use the table to set up more reads from your device. Fill in the **Read Registers** table to read data from two timers and three switches. Since the table automatically displays the register you configure in the **Configure** tab, the first timer is already listed. Change the name to `C1`, then add four more rows so you have these reads set up.

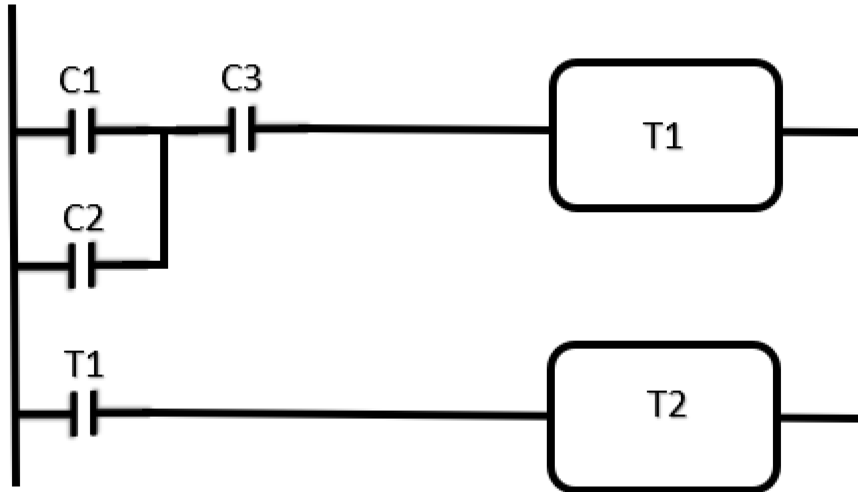
```
Switches
C1, Address 16385, Coil, bit
C2, Address 16386, Coil, bit
C3, Address 16387, Coil, bit
Timers
T1, Address 45057, Holding Register, uint16
T2, Address 45058, Holding Register, uint16
```

The table should look like this:

Read Registers Click 'Resume Reads' to start reading live data. PAUSED Resume Reads

Sel...	Name	Address	Register Type	Precision	Read Value
☐	T2	45058	Holding Register	uint16	
☐	T1	45057	Holding Register	uint16	
☐	C3	16387	Coil	bit	
☐	C2	16386	Coil	bit	
☐	C1	16385	Coil	bit	

The PLC that contains these timers and switches is shown here.



8 To perform the reads on the five registers in the table, click **Resume Reads**.

The **Read Value** column displays the value that is returned and the status indicator changes to `LIVE`, as shown here.

Read Registers Enter register data in the table to read live data. LIVE Resume Reads

Sel...	Name	Address	Register Type	Precision	Read Value
☐	T2	45058	Holding Register	uint16	0
☐	T1	45057	Holding Register	uint16	0
☐	C3	16387	Coil	bit	0
☐	C2	16386	Coil	bit	0
☐	C1	16385	Coil	bit	0

In this case, the value of 0 means the switch or timer is connected and available, but it is not activated.

9 To turn on one of the switches, `C1`, perform a write to the register. In the **Write Registers** section, fill in the following:

Write Registers Write data to a single register.

Address 16385

Register Type Coil

Precision bit

Write Value 1

Write

After you have entered all of the fields, the **Write** button becomes activated.

10 To send the value to the register, click **Write**.

Since you have the same register listed in the **Read Registers** table, you see the read value update when you click **Write**. In the example shown here, you can see that the value of 1 was sent to the register and that it is now reflected in the read table for `C1`, indicating that the switch is turned on.

Read Registers Enter register data in the table to read live data. LIVE Resume Reads

Sel...	Name	Address	Register Type	Precision	Read Value
☐	T2	45058	Holding Register	uint16	0
☐	T1	45057	Holding Register	uint16	0
☐	C3	16387	Coil	bit	0
☐	C2	16386	Coil	bit	0
☐	C1	16385	Coil	bit	1

11 Perform another write to turn the `C3` switch on. In the **Write Registers** section, fill in the following:

```
Address: 16387
Type: Coil
Precision: bit
Write Value: 1
```

Click the **Write** button.

Once that switch is on, the timers turn on, since that is how the PLC board is arranged. `T1` is turned on when the switches are on, and then 5 seconds later `T2` is automatically turned on. At that point both of the timers and two of the switches are turned on, as shown here.

Read Registers Enter register data in the table to read live data. LIVE Resume Reads

Select	Name	Address	Register Type	Precision	Read Value
☐	T2	45058	Holding Register	uint16	2
☐	T1	45057	Holding Register	uint16	7
☐	C3	16387	Coil	bit	1
☐	C2	16386	Coil	bit	0
☐	C1	16385	Coil	bit	1

See Also

Related Examples

- "Configure a Connection in the Modbus Explorer" on page 11-20
- "Read Coils, Inputs, and Registers in the Modbus Explorer" on page 11-23
- "Write to Coils and Holding Registers in the Modbus Explorer" on page 11-26
- "Generate a Script from Your Modbus Explorer Session" on page 11-33

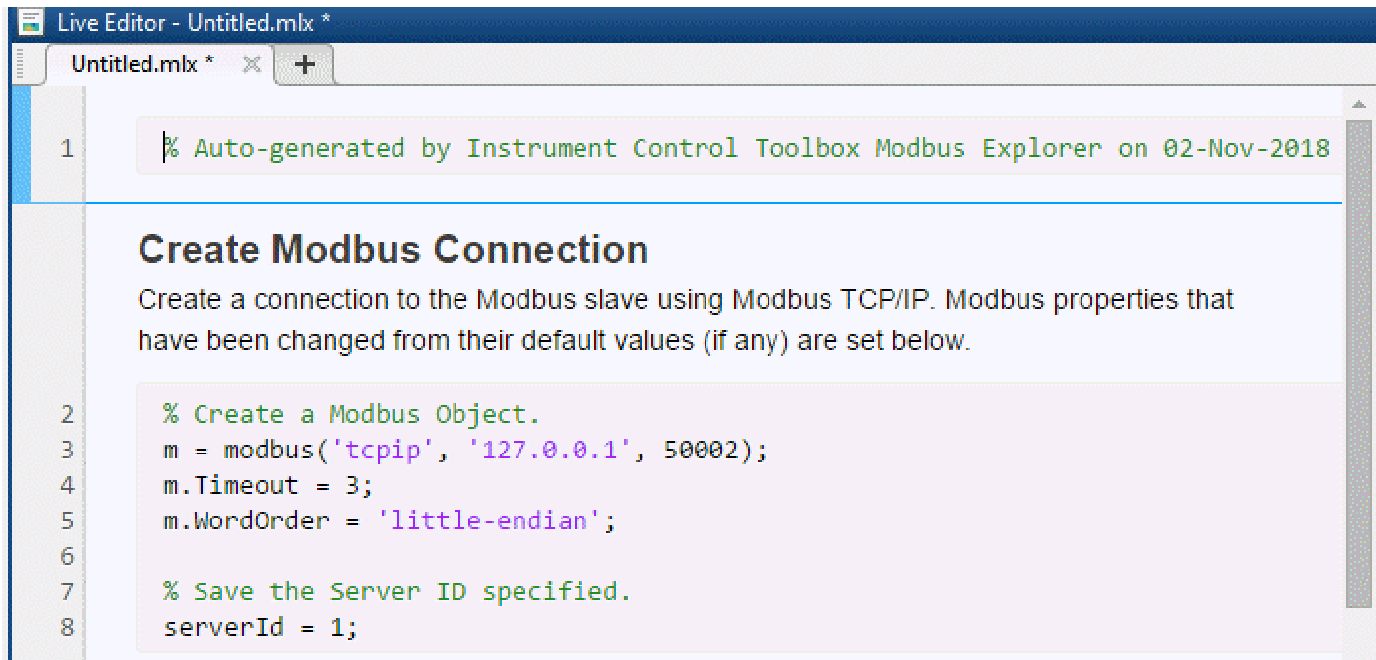
Generate a Script from Your Modbus Explorer Session

You can generate a MATLAB script from your Modbus Explorer session, and then run it at the command line using the Instrument Control Toolbox Modbus functionality. The generated script contains your device configuration, all the read operations that you perform, the last write operation that you perform for each register type, and cleanup tasks.

Note that generating a script is not the same as saving the contents of the **Read Register** table. To do that, use **Export** as described in “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23.

To generate a script from your Modbus Explorer session, click **Generate Script** in the toolstrip. The script appears in the MATLAB Editor as a live script. To keep the generated script, save it in the editor. An example of a script is shown here.

Create Modbus Connection - This section of the generated script creates the `modbus` object using the configuration properties specified.



Perform Modbus Reads - This section of the script performs all the Modbus reads that were done in your session. In this example, five reads were performed.

Perform Modbus Reads

Perform a read on Modbus registers from the Register Table. The read data is stored in a struct 'modbusData'. The name provided for each address of the Register Table is a field of 'modbusData'. The read value for each register is stored in the respective fields of 'modbusData'.

```
9   % Holding Registers
10  % Read 1 Holding Register of type 'uint16' starting from address 10.
11  modbusData.Reg_2 = read(m, 'holdingregs', 10, 1, serverId, 'uint16');
12  
13  % Read 1 Holding Register of type 'uint32' starting from address 2005.
14  modbusData.Reg_5 = read(m, 'holdingregs', 2005, 1, serverId, 'uint32');
15  
16  % Read 1 Holding Register of type 'single' starting from address 6001.
17  modbusData.Reg_1 = read(m, 'holdingregs', 6001, 1, serverId, 'single');
18  
19  % Read 1 Holding Register of type 'double' starting from address 7001.
20  modbusData.Reg_4 = read(m, 'holdingregs', 7001, 1, serverId, 'double');
21  
22  % Input Registers
23  % Read 1 Input Register of type 'uint64' starting from address 4001.
24  modbusData.Reg_3 = read(m, 'inputregs', 4001, 1, serverId, 'uint64');
```

Perform Modbus Writes - This section of the script shows the last Modbus write operation that was done in the session for each register type. It is presented as a comment so that you must intentionally uncomment it to do the write, to prevent unexpected actions on your device.

Perform Modbus Writes (Example)

Perform a write on Modbus registers. The last successful Modbus write for each register type (Coils and Holding Registers) that have been done using Modbus Explorer are shown below as comments.

```
25  % Holding Registers
26  % Write a value 30 of type 'uint32' to a Holding Register at address 2005.
27  % write(m, 'holdingregs', 2005, 30, serverId, 'uint32');
```

Clean Up - This section of the script clears the `modbus` object and releases the Server ID.

Clean Up

Clear the Modbus Explorer session variables.

```
28  % Clear the Modbus Object created.
29  clear m
30
31  % Clear the Server ID.
32  clear serverId
```

See Also

Related Examples

- “Configure a Connection in the Modbus Explorer” on page 11-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 11-26
- “Control a PLC Using the Modbus Explorer” on page 11-28

Using Device Objects

This chapter describes specific features and actions related to using device objects.

- “Device Objects” on page 12-2
- “Creating and Connecting Device Objects” on page 12-5
- “Communicating with Instruments” on page 12-8
- “Device Groups” on page 12-12

Device Objects

In this section...

"Overview" on page 12-2
"What Are Device Objects?" on page 12-2
"Device Objects for MATLAB Instrument Drivers" on page 12-3

Overview

All instruments attached to your computer must communicate through an interface. Popular interface protocols include GPIB, VISA, RS-232 (serial), and RS-485 (serial). While Instrument Control Toolbox interface objects allow you to communicate with your equipment at a low (instrument command) level, Instrument Control Toolbox also allows you to communicate with your equipment without detailed knowledge of how the hardware interface operates.

Programmable devices understand a specific language, sometimes referred to as its command set. One common set is called SCPI (Standard Commands for Programmable Instruments).

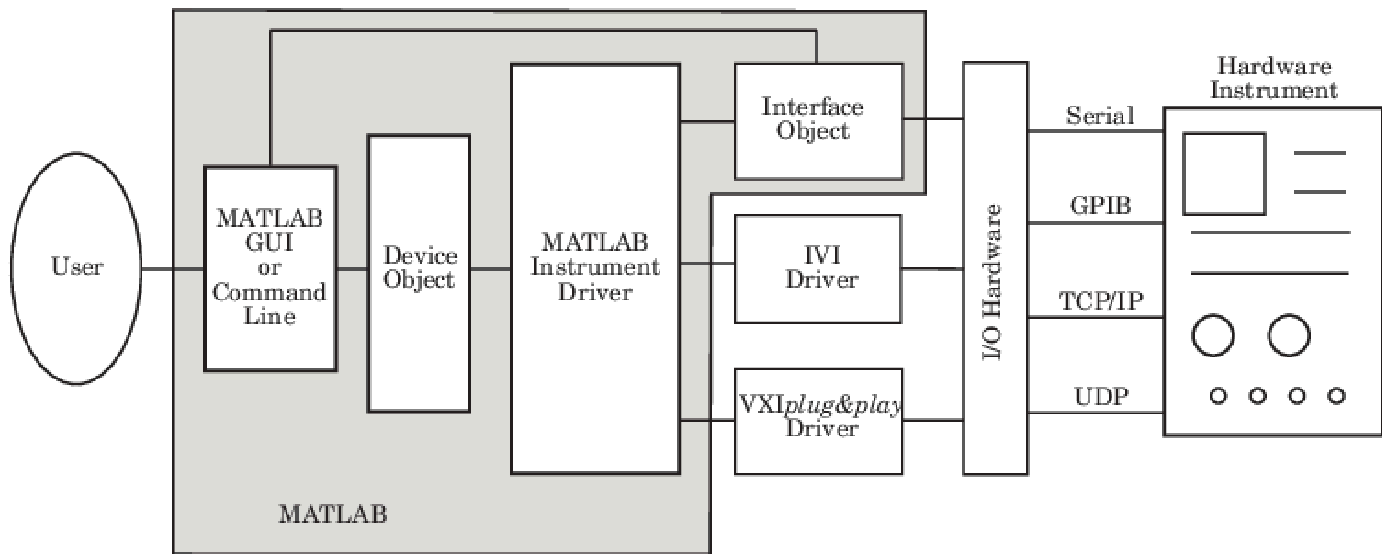
Device objects allow you to configure and query an instrument without knowledge of its command set. This section covers the basic functionality of device objects that use MATLAB instrument drivers.

If your application is straightforward, or if you are already familiar with the topics mentioned above, you might want to begin with "Creating and Connecting Device Objects" on page 12-5. If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to the Getting Started documentation that is linked to from the top of the main Instrument Control Toolbox Doc Center page.

What Are Device Objects?

Device objects are used to represent instruments in MATLAB workspace. Properties and methods specific to an instrument are encapsulated within device objects. Device objects also free you from the specific underlying commands required to communicate with your hardware.

You can use device objects at the MATLAB Command Window, inside functions, scripts, and graphical user interface callbacks. The low-level communication is performed through a MATLAB instrument driver.



Device Objects for MATLAB Instrument Drivers

There are three types of MATLAB instrument drivers:

- MATLAB interface instrument driver
- MATLAB IVI instrument driver
- MATLAB VXI*plug&play* instrument driver
- Generic instrument driver

Instrument Control Toolbox device objects support all these types of MATLAB drivers, so that by using a device object, you can interface with any of these drivers in the same way. However, each of these drivers interfaces differently with the hardware. While MATLAB IVI and MATLAB VXI*plug&play* drivers interface directly through standard drivers and the hardware port to the instrument, the MATLAB interface driver requires an interface object to communicate with the instrument. You can use generic drivers to communicate with devices or software. For more information on generic drivers, see “Generic Drivers: Overview” on page 16-2.

The Instrument Control Toolbox software supports the following interface objects:

- `gpib`
- `serial`
- `tcpip`
- `udp`
- `visa`

To learn how to create and use interface objects, see “Create Interface Object” on page 3-2.

Note If you are using an interface object with a device object and a MATLAB interface driver, you do not need to connect the interface object to the interface using the `fopen` command. You need to connect the device object only.

Available MATLAB Instrument Drivers

Several drivers ship with the Instrument Control Toolbox software. You can find these drivers by looking in the directory

```
matlabroot\toolbox\instrument\instrument\drivers
```

where *matlabroot* is the MATLAB installation directory, as seen when you type

```
matlabroot
```

at the MATLAB Command Window.

Many other drivers are available on the MathWorks Web site at

```
https://www.mathworks.com/matlabcentral/fileexchange
```

including drivers specifically for the Instrument Control Toolbox software.

Creating and Connecting Device Objects

In this section...

"Device Objects for MATLAB Interface Drivers" on page 12-5
"Device Objects for VXIplug&play and IVI Drivers" on page 12-6
"Connecting the Device Object" on page 12-6

Device Objects for MATLAB Interface Drivers

Create a MATLAB device object to communicate with a Tektronix TDS 210 Oscilloscope. To communicate with the scope you will use a National Instruments GPIB controller.

1. First create an interface object for the GPIB hardware. The following command creates a GPIB object for a National Instruments GPIB board at index 0 with an instrument at primary address 1.

   ```
   g = gpib('ni',0,1);
   ```
2. Now that you have created the interface object, you can construct a device object that uses it. The command to use is `icdevice`. You need to supply the name of the instrument driver, `tektronix_tds210`, and the interface object created for the GPIB controller, `g`.

   ```
   d = icdevice('tektronix_tds210', g);
   ```

You can use the `whos` command to display the size and class of `d`.

```
whos d
  Name      Size                    Bytes  Class

  d         1x1                       652  icdevice object

Grand total is 22 elements using 652 bytes
```

Device Object Properties

A device object has a set of base properties and a set of properties defined by the driver. All device objects have the same base properties, regardless of the driver being used. The driver properties are defined by the driver specified in the `icdevice` constructor.

Device Object Display

Device objects provide you with a convenient display that summarizes important object information. You can invoke the display in these ways:

- Type the name of the device object at the command line.
- Exclude the semicolon when creating the device object.
- Exclude the semicolon when configuring properties using dot notation.
- Pass the object to the `disp` or `display` function.

The `display` summary for device object `d` is given below.

```
Instrument Device Object Using Driver : tektronix_tds210.mdd

    Instrument Information
      Type:              Oscilloscope
```

```
      Manufacturer:       Tektronix
      Model:              TDS210

   Driver Information
      DriverType:         MATLAB interface object
      DriverName:         tektronix_tds210.mdd
      DriverVersion:      1.0

   Communication State
      Status:             open
```

You can also display summary information via the Workspace browser by right-clicking a device object and selecting **Display Summary** from the context menu.

Device Objects for VXIplug&play and IVI Drivers

Creating the MATLAB Instrument Driver

The command-line function `makemid` creates a MATLAB instrument driver from a VXI*plug&play* or IVI-C driver, saving the new driver in a file on disk. The syntax is

```
makemid('driver','filename')
```

where `driver` is the original VXI*plug&play* or IVI-C driver name (identified by `instrhwinfo` or the Test & Measurement Tool), and `filename` is the file containing the newly created MATLAB instrument driver. See the `makemid` reference page for a full description of the function and all its options.

You can open the new driver in the MATLAB Instrument Driver Editor, and then modify and save it as required.

Creating the Device Object

After you create the MATLAB instrument driver by conversion, you create the device object with the filename of the new driver as an argument for `icdevice`.

For example, if the driver is created from a VXI*plug&play* or IVI-C driver,

```
obj = icdevice('ConvertedDriver.mdd','GPIB0::2::INSTR')
```

Connecting the Device Object

Now that you have created the device object, you can connect it to the instrument with the `connect` function. To connect the device object, `d`, created in the last example, use the following command:

```
connect(d);
```

By default, the property settings are updated to reflect the current state of the instrument. You can modify the instrument settings to reflect the device object's property values by passing an optional update parameter to `connect`. The update parameter can be either `object` or `instrument`. To have the instrument updated to the object's property values, the `connect` function from the previous example would be

```
connect(d, 'instrument');
```

If `connect` is successful, the device object's status property is set to `open`; otherwise it remains as `closed`. You can check the status of this property with the `get` function or by looking at the object display.

```
d.status

ans =

    open
```

Communicating with Instruments

In this section...

"Configuring Instrument Settings" on page 12-8
"Calling Device Object Methods" on page 12-9
"Control Commands" on page 12-10

Configuring Instrument Settings

Once a device object has been created and connected, it can be used as the interface to an instrument. This chapter shows you how to access and configure your instrument's settings, as well as how to read and write data to the instrument.

Every device object contains properties specific to the instrument it represents. These properties are defined by the instrument driver used during device object creation. For example, there may be properties for an oscilloscope that allow you to adjust trigger parameters, or the contrast of the screen display.

Properties are assigned default values at device object creation. On execution of `connect` the object is updated to reflect the state of the instrument or vice versa, depending on the second argument given to `connect`.

You can obtain a full listing of configurable properties by calling the `set` command and passing the device object.

Configuring Settings on an Oscilloscope

This example illustrates how to configure an instrument using a device object.

The instrument used is a Tektronix TDS 210 two-channel oscilloscope. A square wave is input into channel 1 of the oscilloscope. The task is to adjust the scope's settings so that triggering occurs on the falling edge of the signal:

1. **Create the device object —** Create a GPIB interface object, and then a device object for a TDS 210 oscilloscope.

```
g = gpib('ni',0,1);
d = icdevice('tektronix_tds210', g);
```

2. **Connect the device object** — Use the `connect` function to connect the device object to the instrument.

```
connect(d);
```

3. **Check the current `Slope` settings for the `Trigger` property—** Create a variable to represent the Trigger property and then use the `get` function to obtain the current value for the oscilloscope `Slope` setting.

```
dtrigger = get(d, 'Trigger');
dtrigger.Slope
ans =

    rising
```

   The `Slope` is currently set to `rising`.

4 **Change the `Slope` setting** — If you want triggering to occur on the falling edge, you need to modify that setting in the device object. This can be accomplished with the `set` command.

```
dtrigger.Slope = 'falling');
```

This changes `Slope` to `falling`.

5 **Disconnect and clean up** — When you no longer need the device object, disconnect it from the instrument and remove it from memory. Remove the device object and interface object from the MATLAB workspace.

```
disconnect(d);
delete(d);
clear d g dtrigger;
```

Calling Device Object Methods

Device objects contain methods specific to the instruments they represent. Implementation details are hidden behind a single function. Instrument-specific functions are defined in the MATLAB instrument driver.

The `methods` function displays all available driver-defined functions for the device object. The display is divided into two sections:

- Generic object functions
- Driver-specific object functions

To view the available methods, type

```
methods(obj)
```

Use the `instrhelp` function to get help on the device object functions.

```
instrhelp(obj, methodname);
```

To call instrument-specific methods you use the `invoke` function. `invoke` requires the device object and the name of the function. You must also provide input arguments, when appropriate. The following example demonstrates how to use `invoke` to obtain measurement data from an oscilloscope.

Using Device Object Functions

This example illustrates how to call an instrument-specific device object function. Your task is to obtain the frequency measurement of a waveform. The instrument is a Tektronix TDS 210 two-channel oscilloscope.

The scope has been preconfigured with a square wave input into channel 1 of the oscilloscope. The hardware supports four different measurements: frequency, mean, period, and peak-to-peak. The requested measurement is signified with the use of an index variable from 1 to 4.

For demonstration purposes, the oscilloscope in this example has been preconfigured with the correct measurement settings:

1 **Create the device object** — Create a GPIB interface object and a device object for the oscilloscope.

```
g = gpib('ni',0,1);
d = icdevice('tektronix_tds210', g);
```

2 **Connect the device object** — Use the connect command to open the GPIB object and update the settings in the device object.

```
connect(d);
```

3 **Obtain the frequency measurement** — Use the invoke command and call measure. The measure function requires that an index parameter be specified. The value of the index specifies which measurement the oscilloscope should return. For the current setup of the Tektronix TDS 210 oscilloscope, an index of 1 indicates that frequency is to be measured.

```
invoke(d, 'measure', 1)

ans =

    999.9609
```

The frequency returned is 999.96 Hz, or nearly 1 kHz.

4 **Disconnect and clean up** — You no longer need the device object so you can disconnect it from the instrument. You should also delete it from memory and remove it from the MATLAB workspace.

```
disconnect(d);
delete(d);
clear d g;
```

Control Commands

Control commands are special functions and properties that exist for all device objects. You use control commands to identify an instrument, reset hardware settings, perform diagnostic routines, and retrieve instrument errors. The set of control commands consists of

- “InstrumentModel” on page 12-10
- “devicereset” on page 12-11
- “selftest” on page 12-11
- “geterror” on page 12-11

All control commands are defined within the MATLAB instrument driver for your device.

InstrumentModel

InstrumentModel is a device object property. When queried, the instrument identification command is sent to the instrument.

For example, for a Tektronix TDS 210 oscilloscope,

```
d.InstrumentModel

ans =

TEKTRONIX,TDS 210,0,CF:91.1CT FV:v2.03 TDS2MM:MMV:v1.04
```

devicereset

To restore the factory settings on your instrument, use the `devicereset` function. When `devicereset` is called, the appropriate reset instruction is sent to your instrument.

The command accepts a connected device object and has no output arguments.

```
devicereset(obj);
```

selftest

This command requests that your instrument perform a self-diagnostic. The actual operations performed and output arguments are specific to the instrument your device object is connected to. `selftest` accepts a connected device object as an input argument.

```
result = selftest(obj);
```

geterror

You can retrieve error messages generated by your instrument with the `geterror` function. The returned messages are instrument specific. `geterror` accepts a connected device object as an input argument.

```
msg = geterror(obj);
```

Device Groups

In this section...
“Working with Group Objects” on page 12-12
“Using Device Groups to Access Instrument Data” on page 12-12

Working with Group Objects

Device groups are used to group several related properties. For example, a channel group might contain the input channels of an oscilloscope, and the properties and methods specific to the input channels on the instrument.

MATLAB instrument drivers specify the type and quantity of device groups for device objects.

Group objects can be accessed via the `get` command. For the Tektronix TDS 210 oscilloscope, there is a channel group that contains two group objects. The device property to access a group is always the group name.

```
chans = get(d, 'Channel')

   HwIndex:    HwName:    Type:            Name:
   1           CH1        scope-channel    Channel1
   2           CH2        scope-channel    Channel2
```

To display the functions that a device group object supports, use the `methods` function.

```
methods(chans(1))
```

You can also display a list of the group object's properties and their current settings.

```
chans(2)
```

To get help on a driver-specific property or function, use the `instrhelp` function, with the name of the function or property.

```
instrhelp(chans(1),'Coupling')
```

Using Device Groups to Access Instrument Data

This example shows how to obtain waveform data from a Tektronix TDS 210 oscilloscope with a square wave signal input on channel 1, on a Windows machine. The methods used are specific to this instrument:

1 **Create and connect** — First, create the device object for the oscilloscope and then connect to the instrument.

   ```
   s = serial('com1');
   d = icdevice('tektronix_tds210', s);
   connect(d);
   ```
2 **Get the device group** — To retrieve waveform data, first gain access to the Waveform group for the device object.

   ```
   w = d.waveform;
   ```

This group is specific for the hardware you are using. The TDS 210 oscilloscope has one Waveform; therefore the group contains one group object.

```
HwIndex:    HwName:      Type:              Name:
1           Waveform1    scope-waveform     Waveform1
```

3. **Obtain the waveform** — Now that you have access to the Waveform group objects, you can call the `readwaveform` function to acquire the data. For this example, channel 1 of the oscilloscope is reading the signal. To access this channel, call `readwaveform` on the first channel.

```
wave = invoke(w, 'readwaveform', 'channel1');
```

4. **View the data** — The `wave` variable now contains the waveform data from the oscilloscope. Use the `plot` command to view the data.

```
plot(wave);
```

5. **Disconnect and clean up** — Once the task is done, disconnect the hardware and free the memory used by the objects.

```
disconnect(d)
delete([d s])
clear d, s, w, wave;
```

Using VXI*plug&play* Drivers

This chapter describes the use of VXI*plug&play* drivers for instrument control. The sections are as follows.

- “VXIplug&play Setup” on page 13-2
- “VXIplug&play Drivers” on page 13-3

VXIplug&play Setup

In this section...

"Instrument Control Toolbox Software and VXIplug&play Drivers" on page 13-2
"VISA Setup" on page 13-2
"Other Software Requirements" on page 13-2

Instrument Control Toolbox Software and VXIplug&play Drivers

The Instrument Control Toolbox software can communicate with hardware using VXI*plug&play* drivers. Most often, the instrument manufacturers supply these drivers.

For definitions and specifications of VXI*plug&play* drivers, see the IVI Foundation website.

VISA Setup

A system must have VISA installed in order for VXI*plug&play* drivers to work. The driver installer software might specify certain VISA or other connectivity requirements.

To determine whether your system is properly configured with the necessary version of VISA, at the MATLAB Command window, type:

```
instrhwinfo visa
ans =
    InstalledAdaptors: {'agilent'}
       JarFileVersion: 'Version 2.0 (R14)'
```

The cell array returned for `InstalledAdaptors` indicates which VISA software is installed. A 1x0 cell array indicates that no VISA is installed. Possible `InstalledAdaptors` values are `keysight` (note that `agilent` also still works), `tek`, `rs`, and `ni`.

If you do not have VISA installed, you need to install it. The software installation disk provided with your instrument might include VISA along with the instrument's VXI*plug&play* driver, or you might be able to download VISA from the instrument manufacturer's Web site.

Other Software Requirements

An instrument driver can have other software requirements in addition to or instead of VISA. Consult the driver documentation. The installer software itself might specify these requirements.

VXIplug&play Drivers

In this section...

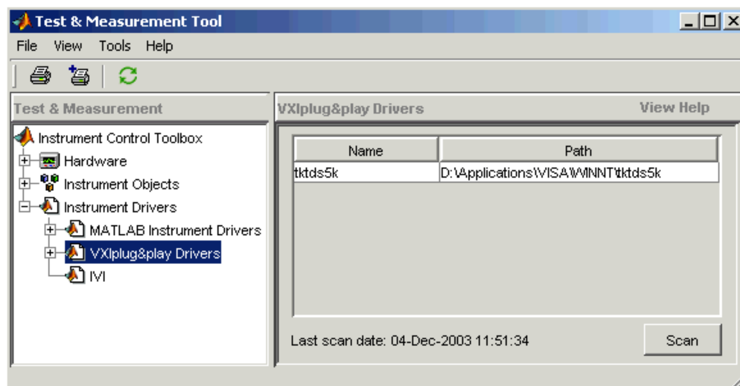
"Installing VXIplug&play Drivers" on page 13-3
"Creating a MATLAB VXIplug&play Instrument Driver" on page 13-3
"Constructing Device Objects Using a MATLAB VXIplug&play Instrument Driver" on page 13-6
"Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI" on page 13-6

Installing VXIplug&play Drivers

The VXI*plug&play* driver particular to a piece of equipment is usually provided by the equipment manufacturer as either an installation disk or as a Web download. Once the driver is installed, you can determine whether the configuration is visible to MATLAB software by using the Test & Measurement Tool to view the current driver installations. Open the tool by typing:

```
tmtool
```

Expand the `Instrument Drivers` node and click `VXIplug&play Drivers`. Click the **Scan** button to update the display. All installed VXI*plug&play* drivers will be listed.



Alternatively, you can use the function `instrhwinfo` to find out which drivers are installed.

```
instrhwinfo ('vxipnp')
ans =
    InstalledDrivers: {'tktds5k', 'ag3325b', 'hpe363xa'}
      VXIPnPRootPath: 'C:\VXIPNP\WINNT'
```

The cell array returned for `InstalledDrivers` contains the names of all the installed VXI*plug&play* drivers. The string returned for `VXIPnPRootPath` indicates where the drivers are installed.

Creating a MATLAB VXIplug&play Instrument Driver

To use a VXI*plug&play* driver with a device object, you must have a MATLAB VXI*plug&play* instrument driver based upon the information in the original VXI*plug&play* driver. The MATLAB VXI*plug&play* instrument driver, whether modified or not, acts as a wrapper to the VXI*plug&play* driver. You can download or create the MATLAB instrument driver.

Downloading a Driver from the MathWorks website

You might find an appropriate MATLAB driver wrapper for your instrument from the VXIplug&play page on the MathWorks website. You can search for the driver you need or you can submit a request to MathWorks for your particular driver.

To use the downloaded MATLAB VXI*plug&play* driver, you must also have the instrument's VXI*plug&play* driver installed. This driver is probably available from the instrument manufacturer's website.

Creating a Driver with makemid

The command-line function `makemid` creates a MATLAB VXI*plug&play* instrument driver from a VXI*plug&play* driver, saving the new driver in a file on disk. The syntax is

```
makemid('driver','filename')
```

where `driver` is the original VXI*plug&play* instrument driver name (identified by `instrhwinfo`), and *`filename`* is the file containing the resulting MATLAB instrument driver. See the `makemid` reference page for details on this function.

If you need to customize the driver, open the new driver in the MATLAB Instrument Driver Editor, modify it as required, and save it.

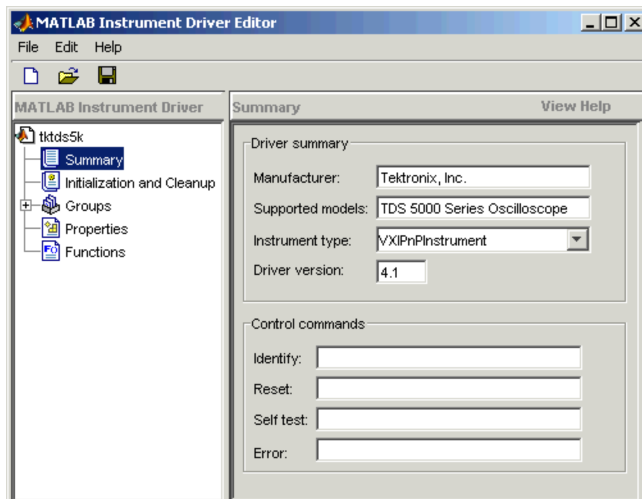
Note When you create a MATLAB instrument driver based on a VXI*plug&play* driver, the original driver must remain installed on your system for you to use the new MATLAB instrument driver.

Importing with the MATLAB Instrument Driver Editor (midedit)

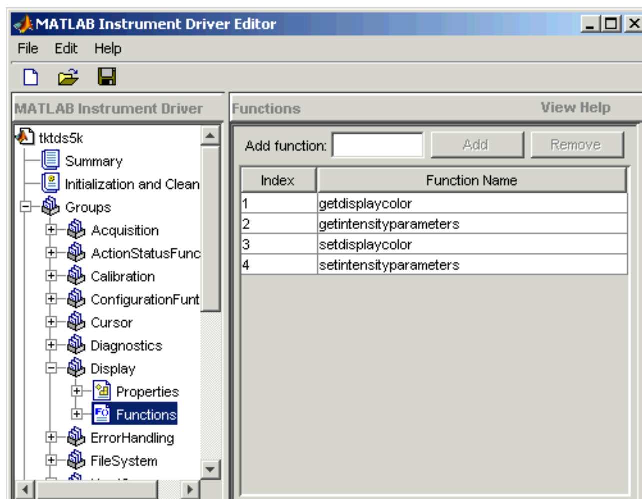
The MATLAB Instrument Driver Editor can import a VXI*plug&play* driver, thereby creating a MATLAB VXI*plug&play* instrument driver. You can evaluate or set the driver's functions and properties, and you can save the modified MATLAB instrument driver for further use:

1. Open the MATLAB Instrument Driver Editor with `midedit`.
2. Select **File > Import**.
3. In the Import Driver dialog box, select the VXI*plug&play* driver that you want to import and click **Import**.

   The MATLAB Instrument Driver Editor loads the driver and displays the components of the driver, as shown in the following figures.



MATLAB Instrument Driver Editor Showing tktds5k MATLAB Instrument Driver Summary



tktds5k MATLAB Instrument Driver Display Group Functions

With the MATLAB Instrument Driver Editor, you can:

- Create, delete, modify, and rename properties, functions, or groups.
- Add code around instrument commands for analysis.
- Add create, connect, and disconnect code.
- Save the driver as a MATLAB VXI*plug&play* instrument driver.

For more information, see “MATLAB Instrument Driver Editor Overview” on page 19-2.

Note When you create a MATLAB instrument driver based on a VXI*plug&play* driver, the original driver must remain installed on your system for you to use the new MATLAB instrument driver.

Constructing Device Objects Using a MATLAB VXIplug&play Instrument Driver

Once you have the MATLAB VXI*plug&play* instrument driver, you create the device object with the file name of the driver and a VISA resource name as arguments for `icdevice`. For example:

```
obj = icdevice('MATLABVXIpnpDriver.mdd','GPIB0::2::INSTR')
connect(obj)
```

See the `icdevice` reference page for full details about this function.

Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI

When using IVI-C or VXI*plug&play* drivers, executing your code will generate additional file(s) in the folder specified by executing the following code at the MATLAB prompt:

```
sprintf('%s',[tempdir 'ICTDeploymentFiles'])
```

On all supported platforms, a file with the name `MATLABPrototypeFor<driverName>.m` is generated, where `<driverName>` the name of the IVI-C or VXI*plug&play* driver. With 64-bit MATLAB on Windows, a second file by the name `<driverName>_thunk_pcwin64.dll` is generated. When creating your deployed application or shared library, manually include these generated files. For more information on including additional files refer to the MATLAB Compiler documentation.

Using IVI Drivers

This chapter describes the use of IVI drivers for instrument control.

- “IVI Drivers Overview” on page 14-2
- “Instrument Interchangeability” on page 14-3
- “Getting Started with IVI Drivers” on page 14-5
- “IVI Configuration Store” on page 14-12
- “Using IVI-C Class-Compliant Wrappers” on page 14-17
- “The Quick-Control Interfaces” on page 14-20
- “Quick-Control Oscilloscope Requirements” on page 14-21
- “Read Waveforms Using the Quick-Control Oscilloscope” on page 14-22
- “Read a Waveform Using a Tektronix Scope” on page 14-24
- “Quick-Control Oscilloscope Functions” on page 14-26
- “Quick-Control Oscilloscope Properties” on page 14-28
- “Quick-Control Function Generator Requirements” on page 14-30
- “Generate Standard Waveforms Using the Quick-Control Function Generator” on page 14-31
- “Generate Arbitrary Waveforms Using Quick-Control Function Generator” on page 14-33
- “Quick-Control Function Generator Functions” on page 14-35
- “Quick-Control Function Generator Properties” on page 14-37
- “Quick-Control RF Signal Generator Requirements” on page 14-40
- “Quick-Control RF Signal Generator Functions” on page 14-41
- “Quick-Control RF Signal Generator Properties” on page 14-43
- “Download and Generate Signals with RF Signal Generator” on page 14-45
- “Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI” on page 14-48

IVI Drivers Overview

In this section...

"Instrument Control Toolbox Software and IVI Drivers" on page 14-2
"IVI-C" on page 14-2

Instrument Control Toolbox Software and IVI Drivers

Instrument Control Toolbox software communicates with instruments using Interchangeable Virtual Instrument (IVI) drivers. The toolbox supports IVI-C drivers, provided by various instrument manufacturers.

For definitions and specifications of IVI drivers and their components, see the IVI Foundation website.

IVI-C

Instrument Control Toolbox software supports IVI-C drivers, with class-compliant and instrument-specific functionality.

IVI class-compliant drivers support common functionality across a family of related instruments. Use class-compliant drivers to access the basic functionality of an instrument, and the ability to swap instruments without changing the code in your application. With an IVI instrument-specific driver or interface, you can access the unique functionality of the instrument. The instrument-specific driver generally does not accommodate instrument substitution.

For IVI-C drivers, you can use IVI-C class drivers and IVI-C specific drivers. Device objects you construct to call IVI-C class drivers offer interchangeability between similar instruments, and work with all instruments consistent with that class driver. Device objects you construct to call IVI-C specific drivers directly generally offer less interchangeability, but provide access to the unique methods and properties of a specific instrument.

Instrument Interchangeability

Minimal Code Changes

With IVI class-compliant drivers, you can exchange instruments with minimal code changes. You can write your code for an instrument from one manufacturer and then swap it for the same type of instrument from another manufacturer. After editing the configuration file that identifies a new instrument, driver, and the hardware address, you can run your code without modifying it.

Effective Use of Interchangeability

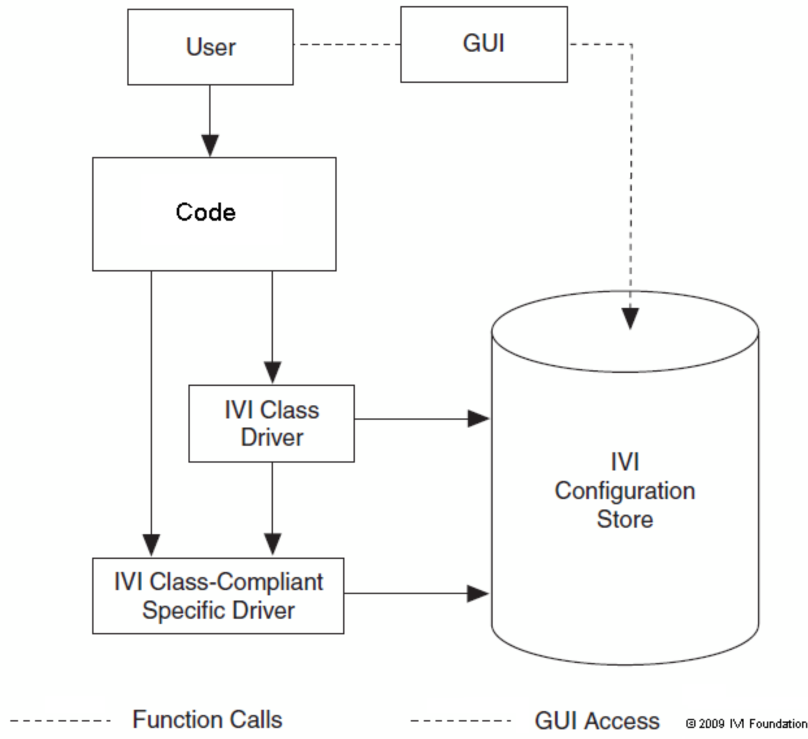
To use the interchangeability of IVI effectively:

- Install drivers for both instruments of the same type (IVI-C).
- Ensure that both drivers implement the same instrument class. For example, both must conform to the requirements for IviDmm or IviScope.
- When using IVI-C your program needs a Class Driver that instantiates the Class Compliant Specific Driver and calls class-compliant functions in it.
- Ensure that your program does not call instrument-specific functions.

You can enhance your code to handle the differences between the instruments or drivers you are using. You can still use these instruments interchangeably.

Examples of Interchangeability

The following diagram show interchangeability between instruments using IVI-C drivers.



Using an IVI-C Class Compliant Driver

Getting Started with IVI Drivers

In this section...
“Introduction” on page 14-5
“Requirements to Work with MATLAB” on page 14-6
“Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI” on page 14-8
“MATLAB IVI Instrument Driver” on page 14-8
“Using MATLAB IVI Wrappers” on page 14-10

Introduction

You need to install IVI drivers and shared components before you can use them in MATLAB. See Requirements on page 14-6 below for more information. You can use an IVI driver in MATLAB in two different ways. The syntax for each method differs vastly. Please refer to the MathWorks IVI Web page for more information. After installing the necessary components, you can:

- Create and use a MATLAB IVI instrument driver as described in MATLAB® IVI Instrument Driver on page 14-8. Here, you create a MATLAB IVI instrument driver with `.mdd` extension using an IVI driver.
- Use a MATLAB IVI wrapper as described in Using MATLAB® IVI Wrappers on page 14-10. Here, MATLAB wraps the IVI driver. You can then use this wrapper with the Instrument Control Toolbox software. This allows interchangeability and is the preferred method if you are working with class-compliant drivers.

  You can use the MATLAB IVI Wrappers provided with the Instrument Control Toolbox software with IVI drivers of the same class. Supported IVI driver classes are:

  - `IviACPwr`
  - `IviCounter`
  - `IviDCPwr`
  - `IviDigitizer`
  - `IviDmm`
  - `IviDownconverter`
  - `IviFgen`
  - `IviPwrMeter`
  - `IviUpconverter`
  - `IviRFSigGen`
  - `IviScope`
  - `IviSpecAn`
  - `IviSwtch`

  You can also use MATLAB IVI wrappers provided by an instrument vendor that has built in MATLAB support. Refer to the vendor documentation for more information about using these drivers in MATLAB.

With the MATLAB IVI instrument driver, you construct a device object, which you use to communicate with your instrument. With the MATLAB IVI wrapper, you communicate with the instrument by directly accessing elements of the driver class.

Requirements to Work with MATLAB

Before you use IVI drivers in MATLAB, install:

- VISA
- IVI Shared components
- Required IVI drivers

Verifying VISA

Most IVI drivers require you to install VISA libraries on your system. The driver installer software specifies certain VISA or other connectivity requirements.

To determine proper configuration of the necessary version of VISA on your system, at the MATLAB Command Window, type:

```
instrhwinfo visa
ans =
    InstalledAdaptors: {'keysight'}
       JarFileVersion: 'Version 2.8.0''
```

The cell array returned for `InstalledAdaptors` indicates the type of VISA software installed. A 1-by-0 cell array indicates that your system does not have VISA installed. Possible `InstalledAdaptors` values are `keysight` (note that `agilent` also still works), `tek`, `rs`, and `ni`.

To install VISA, check the software installation disk provided with your instrument. This disk can include VISA along with the IVI driver for the instrument. You can also download VISA from the website of the instrument manufacturer.

An instrument driver can have other software requirements in addition to or instead of VISA. Consult the driver documentation. The installer software itself can specify these requirements.

Verifying IVI Shared Components

Many driver elements are common to a wide variety of instruments and not contained in the driver itself. You install them separately as *shared components*. Sharing components keeps the drivers as small and interchangeable as possible. You can use `instrhwinfo` to determine whether you installed shared components on your system.

```
instrhwinfo ('ivi')
ans =
.
.
.
ConfigurationServerVersion: '1.6.0.10124'
  MasterConfigurationStore: 'C:\Program Files\IVI\Data\IviConfigurationStore.xml'
               IVIRootPath: 'C:\Program Files\IVI\'
```

`ConfigurationServerVersion`, `MasterConfigurationStore`, and `IVIRootPath` all convey information related to installed shared components. `ConfigurationServerVersion` indicates whether you installed IVI shared components. If its value is an empty character vector, then you have not installed shared components.

Verifying IVI Drivers

The instrument manufacturer usually provides the specific IVI driver, either on an installation disk or as a Web download. Required VISA software and IVI shared components could also come with the driver.

You can use `instrhwinfo` to find information on installed IVI drivers and shared components.

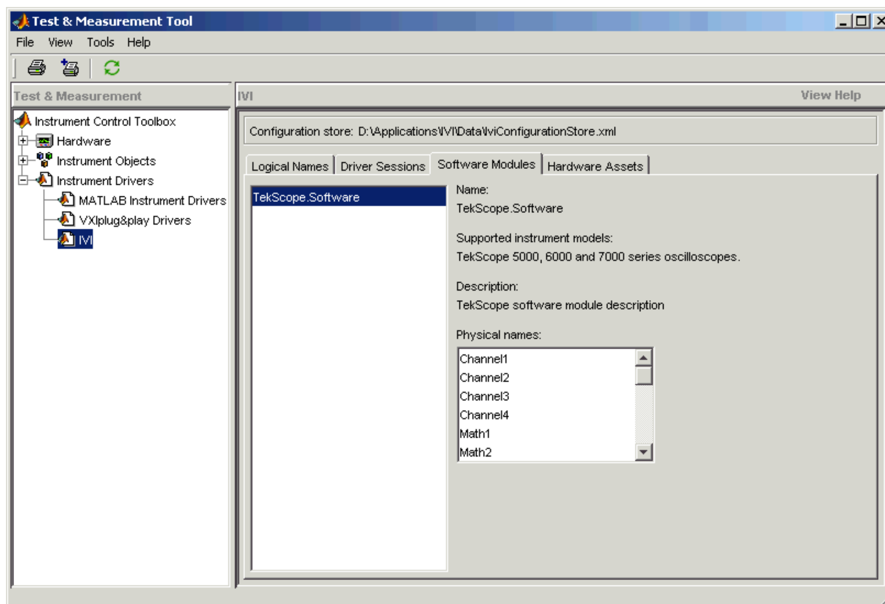
```
instrhwinfo ('ivi')
ans =
              LogicalNames: {'MainScope', 'FuncGen'}
                ProgramIDs: {'TekScope.TekScope','Agilent33250'}
                   Modules: {'ag3325b', 'hpe363xa'}
ConfigurationServerVersion: '1.6.0.10124'
  MasterConfigurationStore: 'C:\Program Files\IVI\Data\
                             IviConfigurationStore.xml'
               IVIRootPath: 'C:\Program Files\IVI\'
```

Logical names are associated with particular IVI drivers, but they do not necessarily imply that the drivers are currently installed. You can install drivers that do not have a `LogicalName` property set yet, or drivers whose `LogicalName` was removed.

Alternatively, use the Test & Measurement Tool to view the installation of IVI drivers and the setup of the IVI configuration store. Open the tool by typing:

```
tmtool
```

Expand the `Instrument Drivers` node and click **IVI**. Click the **Software Modules** tab. (For information on the other IVI driver tabs and settings in the Test & Measurement Tool, see “IVI Configuration Store” on page 14-12.)



Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI

When using IVI-C or VXI*plug&play* drivers, executing your code will generate additional files in the folder specified by executing the following code at the MATLAB prompt:

```
fullfile(tempdir,'ICTDeploymentFiles',sprintf('R%s',version('-release')))
```

On all supported platforms, a file with the name MATLABPrototypeFor<driverName>.m is generated, where <driverName> the name of the IVI-C or VXI*plug&play* driver. With 64-bit MATLAB on Windows, a second file by the name <driverName>_thunk_pcwin64.dll is generated. When creating your deployed application or shared library, manually include these generated files. For more information on including additional files refer to the MATLAB Compiler documentation.

MATLAB IVI Instrument Driver

- "Using a MATLAB IVI Instrument Driver" on page 14-8
- "Creating a MATLAB IVI Instrument Driver with makemid" on page 14-8
- "Downloading a MATLAB IVI Instrument Driver" on page 14-9
- "Importing MATLAB IVI Instrument Drivers" on page 14-9
- "Constructing Device Objects Using a MATLAB IVI Instrument Driver" on page 14-10

Using a MATLAB IVI Instrument Driver

To use an IVI driver with a device object, you need a MATLAB IVI instrument driver based upon the information in the original IVI driver. The MATLAB IVI instrument driver, whether modified or not, acts as a wrapper to the IVI driver. These drivers, however, do not support interchangeability. You can download or create the MATLAB IVI instrument driver.

Creating a MATLAB IVI Instrument Driver with makemid

The command-line function makemid creates a MATLAB IVI instrument driver from an IVI driver, saving the new driver in a file on disk. The syntax is:

```
makemid('driver','filename')
```

driver is the original IVI driver name (identified by instrhwinfo or the Test & Measurement Tool), and *filename* is the MATLAB IVI instrument driver name. For driver use a Module name, a ProgramID, or a LogicalNames value. See the makemid reference page for full details on this function.

To customize the driver, open the new driver in the MATLAB Instrument Driver Editor, modify it as required, and save it.

Tip Do not uninstall the original IVI driver when you create a MATLAB IVI instrument driver based on an IVI driver. You need the IVI driver in order to use the new MATLAB IVI instrument driver.

Note When you create a MATLAB IVI instrument driver without specifying an interface name, makemid uses the instrument-specific interface as the default interface.

Downloading a MATLAB IVI Instrument Driver

Go to the MATLAB Central website and search for an appropriate MATLAB IVI instrument driver for your instrument. You can look for wrappers using the **instrument drivers** tag in the File Exchange area.

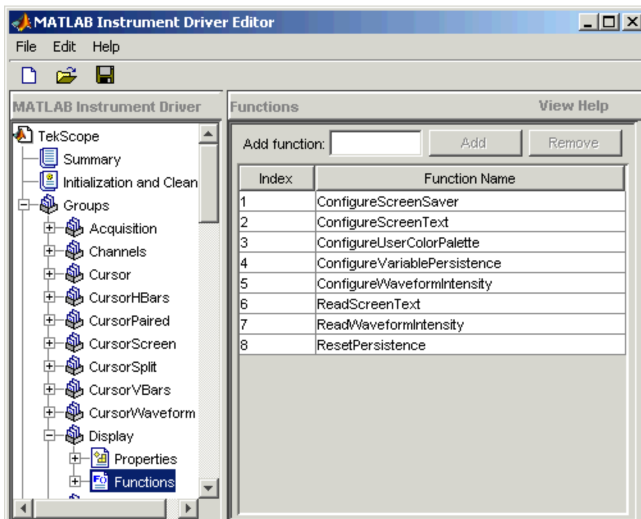
To use the downloaded MATLAB IVI instrument driver, you also need the IVI driver for the installed instrument. Find this driver on the website of the instrument manufacturer.

Importing MATLAB IVI Instrument Drivers

You can import an IVI driver using the MATLAB Instrument Driver Editor, and create a MATLAB IVI instrument driver. Evaluate or set the functions and properties of the driver, and save the modified MATLAB IVI instrument driver for further use.

1. Open the MATLAB Instrument Driver Editor by typing `midedit`.
2. Select **File > Import**. The Import Driver dialog box opens.
3. Select the IVI driver that you want to import, and click **Import**.

   The MATLAB Instrument Driver Editor loads the driver and displays its components.



With the MATLAB Instrument Driver Editor, you can do the following:

- Create, delete, modify, and rename properties, functions, or groups.
- Add code around instrument commands for analysis.
- Add, create, connect, and disconnect code.
- Save the driver as a MATLAB IVI instrument driver.

For more information, see “MATLAB Instrument Driver Editor Overview” on page 19-2.

Tip Do not uninstall the original IVI driver when you create a MATLAB IVI instrument driver based on an IVI driver. You need the IVI driver in order to use the new MATLAB IVI instrument driver.

Constructing Device Objects Using a MATLAB IVI Instrument Driver

Once you have the MATLAB IVI instrument driver, create the device object with the file name of the MATLAB IVI instrument driver as an argument for `icdevice`. The following examples show the creation of the MATLAB IVI instrument driver (all with `.mdd` extensions) and the construction of device objects to use them.

See the `icdevice` and `makemid` reference pages for full details on these functions.

In the following example, `makemid` uses a `LogicalNames` value to identify an IVI driver, then creates a MATLAB IVI instrument driver. Because `LogicalNames` is associated with a driver session and hardware asset, you do not need to pass a `RsrcName` to `icdevice` when constructing the device object.

```
makemid('MainScope','MainScope.mdd');
obj = icdevice('MainScope.mdd');
```

In the next example, `makemid` uses a `ProgramID` to reference an IVI driver, then creates a MATLAB IVI instrument driver. The device object requires a `RsrcName` in addition to the file name of the MATLAB IVI instrument driver.

```
makemid('TekScope.TekScope','TekScopeML.mdd');
obj = icdevice('TekScopeML.mdd','GPIB0::13::INSTR');
```

In the next example, `makemid` uses a software `Module` to reference an IVI-C driver, then creates a MATLAB IVI instrument driver. The device object requires a `RsrcName` in addition to the file name of the MATLAB IVI instrument driver.

```
makemid('ag3325b','Ag3325bML.mdd');
obj = icdevice('Ag3325bML.mdd','ASRL1::INSTR');
```

In the next example, `makemid` creates a MATLAB IVI instrument driver based on the IVI-C class driver `ivifgen`. The device object uses the MATLAB IVI instrument driver file name and the logical name of the driver from the IVI configuration store.

```
makemid('ivifgen','FgenML.mdd');
obj = icdevice('FgenML.mdd','FuncGen');
```

Using MATLAB IVI Wrappers

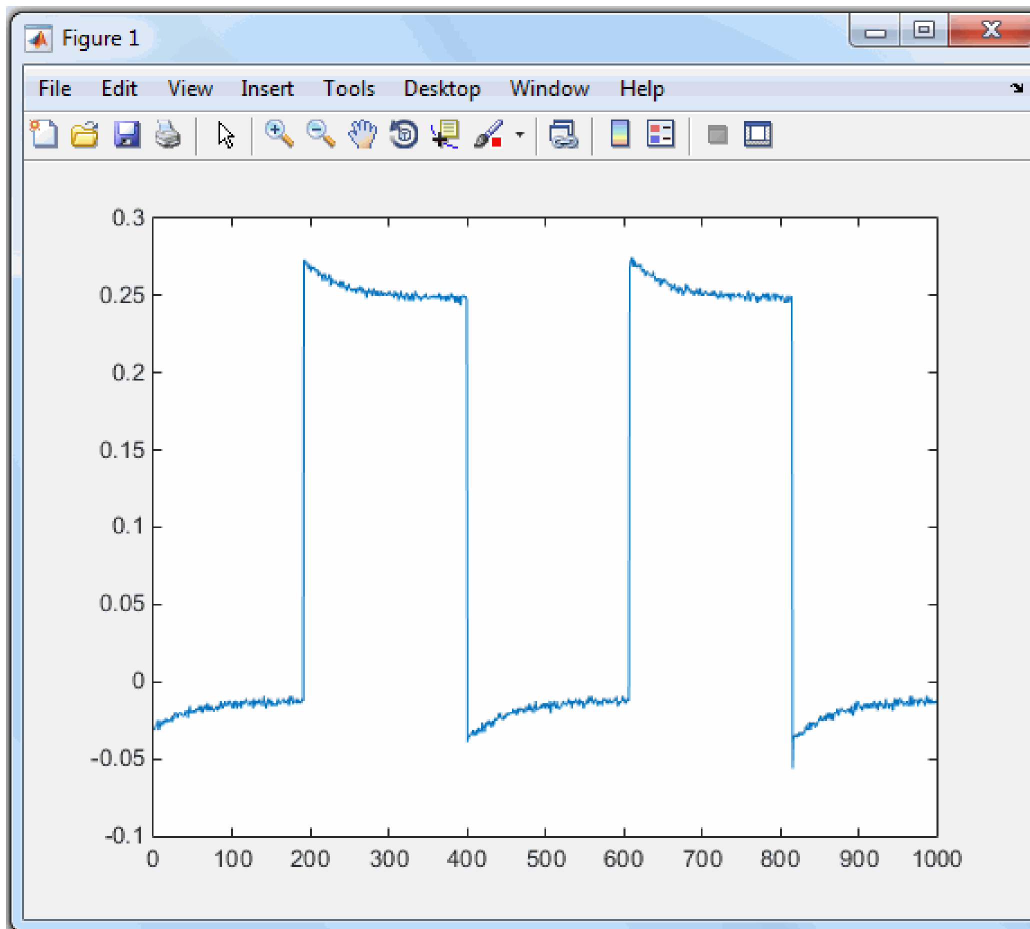
MATLAB IVI wrappers work well with class-compliant drivers.

This example shows how to connect to an instrument and read a waveform using a MATLAB IVI Wrapper.

The instrument in this example is a Keysight MSO6014 mixed signal oscilloscope, with an Agilent546XX driver.

```
%Create the object
myScope = instrument.ivicom.IviScope('Agilent546XX.Agilent546XX');

%Connect to the instrument using the VISA resource string
myScope.Initialize('TCPIP0::xxx-xxxx.xxx.<yourdomain.com>::inst0::INSTR',false,
false,'simulate=false');

%Access the Measeurements Collection
myScopeMeasurements  = myScope.Measurements

%Configure measurement 1
myScopeMeasurements.AutoSetup;
```

```
name = myScopeMeasurements.Name(1);
myScopeMeasurement1 = myScopeMeasurements.Item(name);

%Access the Channels collection
myScopeChannels = myScope.Channels;

%Configure channel 1
name = myScopeChannels.Name(1);
myScopeChannel1=  myScopeChannels.Item(name)
myScopeChannel1.Enabled = 1;

%Configure a trigger
myScope.Trigger.Source = 'Channel1';
myScope.Trigger.Level = 1.0;
myScope.Trigger.Edge.Slope = 'IviScopeTriggerSlopePositive';

%Start the measurement and get the data
myScopeMeasurements.Initiate;
myWaveform = myScopeMeasurement1.FetchWaveform;

%Plot the data
plot(myWaveform);

%Close and delete the object
myScope.Close;
myScope.delete
```



Plot the Waveform Read Using the MATLAB IVI Wrapper

IVI Configuration Store

In this section...
“Benefits of an IVI Configuration Store” on page 14-12
“Components of an IVI Configuration Store” on page 14-12
“Configuring an IVI Configuration Store” on page 14-13

Benefits of an IVI Configuration Store

By providing a way to configure the relationship between drivers and I/O references, an IVI configuration store greatly enhances instrument interchangeability.

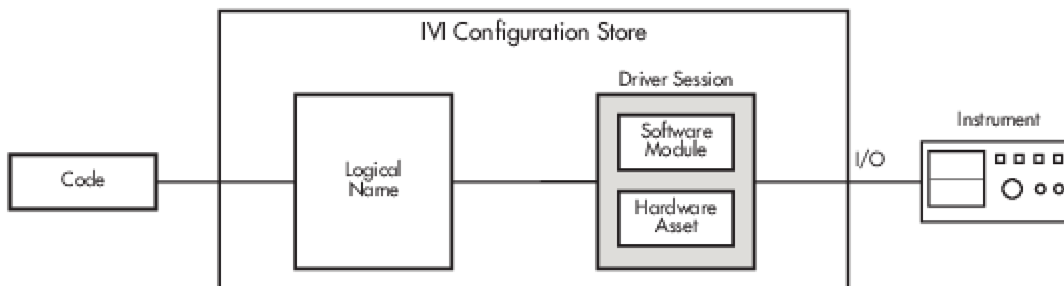
Suppose your code uses only a specified driver to communicate with one type of instrument at a fixed location. If you change the instrument model, instrument location, or driver, you would have to modify the code to accommodate that change.

An IVI configuration store offers the ability to accommodate different instrument models, drivers, or ports, without having to modify your code. This interchangeability is especially useful when you use code that cannot be easily modified.

Components of an IVI Configuration Store

The components of an IVI configuration store identify:

- Locations of the instruments to communicate with
- Software modules used to control the instruments
- Associations of software modules used with instruments at specific locations



Component	Description
Software module	A software module is instrument-specific, and contains the commands and functions necessary to communicate with the instrument. The instrument vendor commonly provides software modules, which you cannot edit from the MATLAB Command Window.
Hardware asset	A hardware asset identifies a communication port connected the instrument. Configure this component with an `IOResourceDescriptor`. Usually you have one hardware asset per connection location (protocol type, bus address, and so on).

Component	Description
Driver session	A driver session makes the association between a software module and a hardware asset. Generally, you have a driver session for each instrument at each of its possible locations. Identical instruments connected at different locations can use the same software module, but because they have different hardware assets, they require different driver sessions. Different kinds of instruments connected to the same location (at different times) use the same hardware asset, but can have different software modules. Therefore, they require different driver sessions.
Logical name	A logical name is a configuration store component that provides access to a driver session. You can interpret a logical name as a configurable pointer to a driver session. In a typical setup, the code communicates with an instrument via a logical name. If the code must communicate with a different instrument (for example, a similar scope at a different location), update only the logical name within the IVI configuration store to point to the new driver session. You need not rewrite any code because it uses the same logical name.

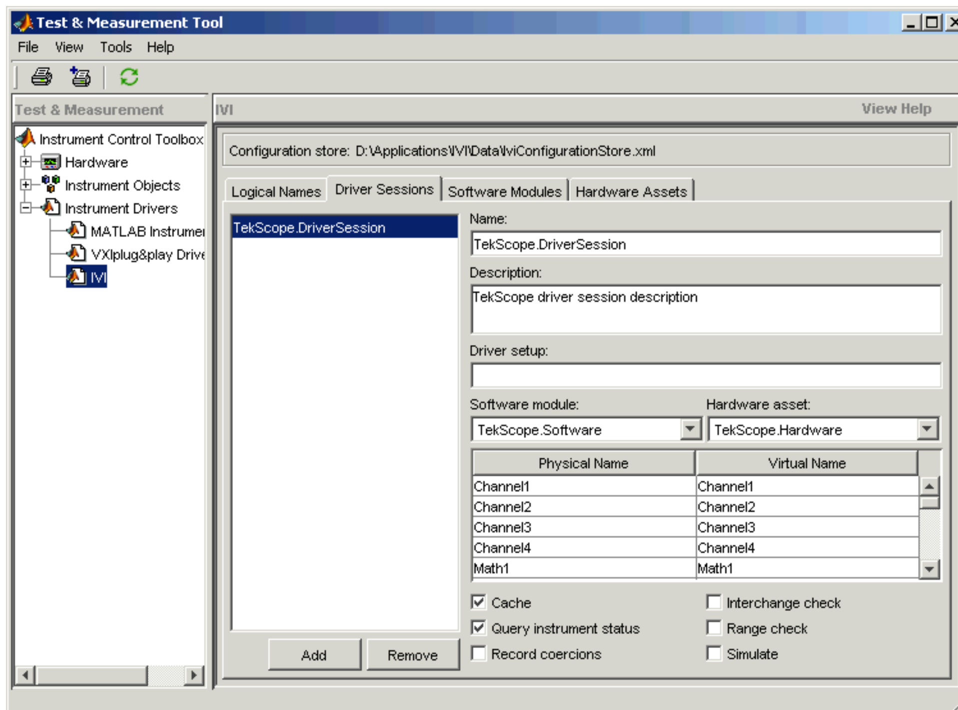
Configuring an IVI Configuration Store

Using the GUI

You can use the Test & Measurement Tool to examine or configure your IVI configuration store. Open the tool by typing:

```
tmtool
```

Expand the **Instrument Drivers** node and click **IVI**.



You see a tab for each type of IVI configuration store element. This figure shows the available driver sessions in the current IVI configuration store. For the selected driver session, you can use any available software module or hardware asset. This figure shows the configuration for the driver session `TekScope.DriverSession`, which uses the software module `TekScope.Software` and the hardware asset `TekScope.Hardware`.

Using the Command Line

Alternatively, you can use command-line functions to examine and configure your IVI configuration store. To see what IVI configuration store elements are available, use `instrhwinfo` to identify the existing logical names.

```
instrhwinfo('ivi')
ans =
              LogicalNames: {'MainScope', 'FuncGen'}
                ProgramIDs: {'TekScope.TekScope','Agilent33250'}
                   Modules: {'ag3325b', 'hpe363xa'}
ConfigurationServerVersion: '1.6.0.10124'
  MasterConfigurationStore: 'C:\Program Files\IVI\Data\
                             IviConfigurationStore.xml'
               IVIRootPath: 'C:\Program Files\IVI\'
```

Use `instrhwinfo` with a logical name as an argument to see the details of the configuration.

```
instrhwinfo('ivi','MainScope')
ans =
            DriverSession: 'TekScope.DriverSession'
            HardwareAsset: 'TekScope.Hardware'
           SoftwareModule: 'TekScope.Software'
     IOResourceDescriptor: 'GPIB0::13::INSTR'
SupportedInstrumentModels: 'TekScope 5000, 6000 and 7000 series'
```

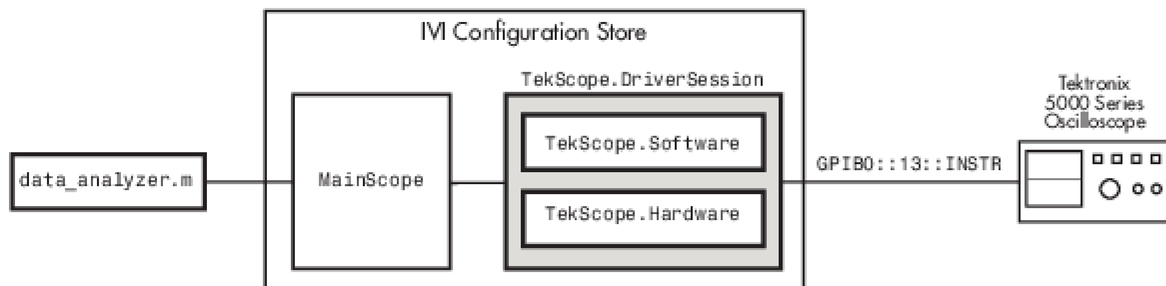
```
        ModuleDescription: 'TekScope software module desc'
           ModuleLocation: ''
```

You can use the command line to change the configuration store. Here is an example of changing it programmatically.

```
% Construct a configStore.
configStore = iviconfigurationstore;

% Set up the hardware asset with name myScopeHWAsset, and resource descriptor
%    TCPIP0::a-m6104a-004598::INSTR.
add(configStore, 'HardwareAsset', 'myScopeHWAsset', 'TCPIP0::a-m6104a-004598::INSTR');

% Add a driver session with name myScopeSession, and use the asset created in the step above.
%    Ag546XX is the Agilent driver.
add(configStore, 'DriverSession', 'myScopeSession', 'Ag546XX', 'myScopeHWAsset');

% Add a logical name to the configStore, with name myScope and driver session
%    named myScopeSession.
add(configStore, 'LogicalName', 'myScope', 'myScopeSession');

% Save the changes to the IVI configuration store data file.
commit(configStore);

% You can verify that the steps you just performed worked.
logicalNameInfo = instrhwinfo('ivi', 'myscope')
```

Basic IVI Configuration Store

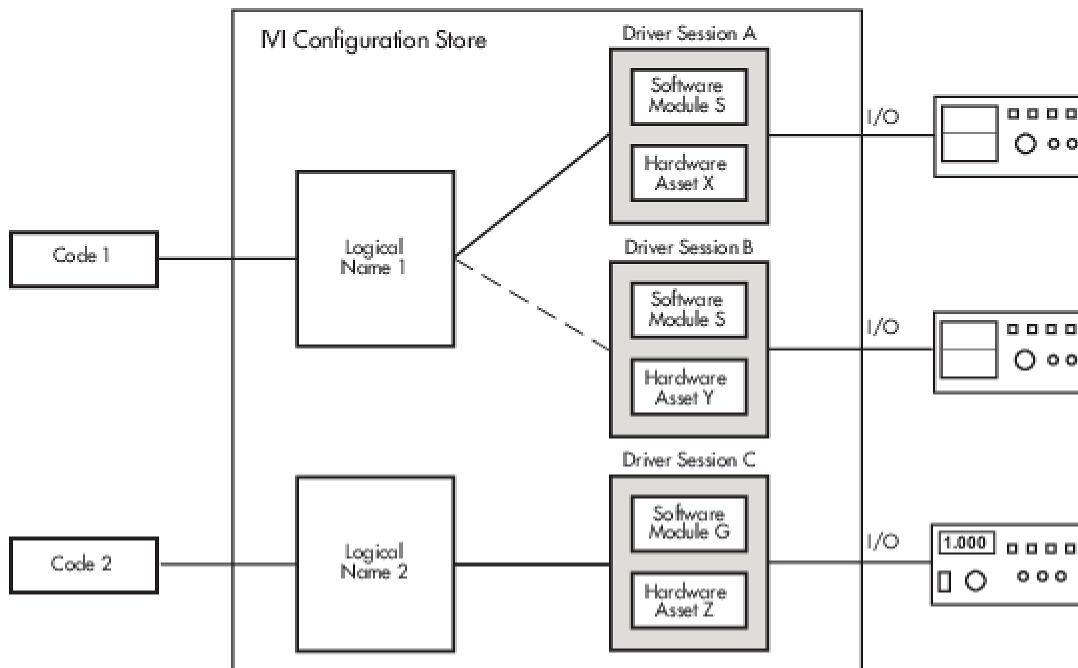
Following is an example of configuration used by `data_analyzer.m`.



Create and configure elements in the IVI configuration store using the IVI configuration store object methods `add`, `commit`, `remove`, and `update`. For further details, see the reference pages for these methods.

IVI Configuration Store with Several Interchangeable Elements

The following figure shows an example of an IVI configuration store with several interchangeable components. **Code 1** requires access to the oscilloscopes at two different locations (hardware asset `X` and hardware asset `Y`). The scopes are similar, so they use the same software module `S`. Here, the scopes are at different locations (or the same scope connected to two different locations at different times). Therefore, each configuration requires its own driver session, in this example, driver session `A` and driver session `B`.



Write **Code 1** to access logical name 1. You configure the name in the IVI configuration store to access driver session A or driver session B (but not both at the same time). Because you select the driver session in the IVI configuration store, you need not alter the code to change access from one scope to the other.

Using IVI-C Class-Compliant Wrappers

In this section...

"IVI-C Wrappers" on page 14-17

"Prerequisites" on page 14-17

"Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI" on page 14-17

"Reading Waveforms Using the IVI-C Class Compliant Interface" on page 14-18

"IVI-C Class Compliant Wrappers in Test & Measurement Tool" on page 14-19

IVI-C Wrappers

The IVI-C wrappers provide an interface to MATLAB for instruments running on IVI-C class-compliant drivers.

This documentation example uses a specific instrument, a Keysight MSO6104A oscilloscope. This feature works with any IVI-C class-compliant instrument. You can follow the basic steps, using your particular instrument if the device is IVI-C class-compliant.

Prerequisites

To use the wrapper you must have the following software installed.

- Windows 64-bit
- VISA shared components
- VISA

  The following example uses Keysight VISA, but you can use any version of VISA.
- National Instruments compliance package NICP 4.1
- Your instrument driver

You can use `instrhwinfo` to confirm that these software modules are installed.

```
% Check that the software is properly installed.
instrhwinfo('ivi')
```

Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI

When using IVI-C or VXI*plug&play* drivers, executing your code will generate additional files in the folder specified by executing the following code at the MATLAB prompt:

```
fullfile(tempdir,'ICTDeploymentFiles',sprintf('R%s',version('-release')))
```

On all supported platforms, a file with the name `MATLABPrototypeFor<driverName>.m` is generated, where `<driverName>` the name of the IVI-C or VXI*plug&play* driver. With 64-bit MATLAB on Windows, a second file by the name `<driverName>_thunk_pcwin64.dll` is generated. When creating your deployed application or shared library, manually include these generated files. For more information on including additional files refer to the MATLAB Compiler documentation.

Reading Waveforms Using the IVI-C Class Compliant Interface

This example shows the general workflow to use with an IVI-C class-compliant device. This example uses a specific instrument, a Keysight MSO6104A oscilloscope. This feature works with any IVI-C class-compliant instrument. You can follow the basic steps using your particular instrument if it is IVI-C class-compliant.

1 Ensure all necessary software is installed. See “Prerequisites” on page 14-17 for the list.

2 Ensure that your instrument is recognized by the VISA utility. In this case, open Keysight Connectivity Expert and make sure it recognizes the oscilloscope.

3 Set up the logical name using the Configuration Store. The VISA resource string shown in this code was acquired from the VISA utility in step 2.

```
% Construct a configStore.
configStore = iviconfigurationstore;

% Set up the hardware asset called myScopeHWAsset, and resource description
     TCPIP0::a-m6104a-004598::INSTR.
add(configStore, 'HardwareAsset', 'myScopeHWAsset', 'TCPIP0::a-m6104a-004598::INSTR');

% Add a driver session called myScopeSession, and use the asset created in the
     step above. Ag546XX is the Agilent driver version.
add(configStore, 'DriverSession', 'myScopeSession', 'Ag546XX', 'myScopeHWAsset');

% Add a logical name to the configStore called myScope and driver session called
     myScopeSession.
add(configStore, 'LogicalName', 'myScope', 'myScopeSession');

% Save the changes to the IVI configuration store data file.
commit(configStore);

% You can verify that the steps you just performed worked.
logicalNameInfo = instrhwinfo('ivi', 'myscope')
```

For more information about the configuration store, see “IVI Configuration Store” on page 14-12.

4 Create an instance of the scope.

```
% Instantiate an instance of the scope.
ivicScope = instrument.ivic.IviScope();
```

5 Connect to the instrument.

```
% Open the hardware session.
ivicScope.init('myScope', true, true);
```

6 Communicate with the instrument. For example, read a waveform.

```
% Use the AutoSetup method to automatically set up the oscilloscope.
ivicScope.Configuration.AutoSetup();

% Create a record length variable.
recordLength = ivicScope.Acquisition.Horizontal_Record_Length;

% Preallocate buffer to store the data read from the scope.
waveformArray = zeros(1, recordLength);

% Read a waveform with channel name set to channel1 and timeout to 1000.
[waveformArray,actualPoints,initiaX,xIncrement] = ivicScope.WaveformAcquisition.
     ReadWaveform('channel1', recordLength, 1000, waveformArray);

% Plot the waveform and assign labels for the plot.
plot(waveformArray);
xlabel('Samples');
ylabel('Voltage');
```

7. After configuring the instrument and retrieving its data, close the session and remove it from the workspace.

   ```
   ivicScope.close();
   clear ivicScope;
   ```

IVI-C Class Compliant Wrappers in Test & Measurement Tool

You can also use the IVI-C Wrappers functionality from the Test & Measurement Tool. View the IVI-C nodes by setting a preference in MATLAB.

1. In MATLAB, on the **Home** tab, in the **Environment** section, click **Preferences**. Then select **Instrument Control** in the Preferences dialog box.
2. Select the **Show IVI Instruments in TMTool** option in the **IVI Instruments** section.

   If you do not have the required software installed, you will get a message indicating that. See “Prerequisites” on page 14-17 for the list of required software.
3. Start the Test & Measurement Tool (using the `tmtool` function), and the new **IVI Instruments** node appears under **Instrument Drivers**.

For information on using it in the Test & Measurement Tool, see the Help within the tool by selecting the **IVI Instruments** node in the tree once it is visible after setting the MATLAB preference.

The Quick-Control Interfaces

The Quick-Control interfaces are used to control oscilloscopes, function generators, or RF signal generators that use an underlying IVI-C driver. You do not have to deal directly with the driver in these easy-to-use interfaces.

There are three Quick-Control interfaces:

- Quick-Control Oscilloscope
- Quick-Control Function Generator
- Quick-Control RF Signal Generator

The Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces contains many of the components required for these interfaces. However, you can install the components independent of the support package. See the following for requirements for each interface.

- "Quick-Control Oscilloscope Requirements" on page 14-21
- "Quick-Control Function Generator Requirements" on page 14-30
- "Quick-Control RF Signal Generator Requirements" on page 14-40

Quick-Control Oscilloscope Requirements

You can use the Quick-Control Oscilloscope for any oscilloscope that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. You can also use it for Tektronix oscilloscopes. This oscilloscope object is an easy to use interface.

The documentation example uses a specific instrument, a Keysight MSO6104 oscilloscope. This feature works with any IVI-C class oscilloscope. You can follow the basic steps, using your particular instrument.

To use the Quick-Control Oscilloscope for an IVI-C scope, you must have the following software installed. Most components are installed by the Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces. To install the support package, see "Install the National Instruments VISA and ICP Interfaces Support Package" on page 15-11.

- Windows 64-bit platforms
- VISA shared components (installed by the support package)
- VISA (installed by the support package)

  Note, the examples use Keysight VISA, but you can use any version of VISA.
- National Instruments IVI compliance package NICP 4.1 or later (installed by the support package)
- Your instrument's device-specific driver. If you do not already have it, go to your instrument vendor's website and download the IVI-C driver for your specific instrument.

You can use `instrhwinfo` to confirm that the required software is installed.

```
% Check that the software is properly installed.
instrhwinfo('ivi')
```

See Also

Related Examples

- "Read Waveforms Using the Quick-Control Oscilloscope" on page 14-22
- "Read a Waveform Using a Tektronix Scope" on page 14-24
- "Quick-Control Oscilloscope Functions" on page 14-26
- "Quick-Control Oscilloscope Properties" on page 14-28

Read Waveforms Using the Quick-Control Oscilloscope

This example shows the general workflow to use for the Quick-Control Oscilloscope. This example uses a specific instrument, a Keysight MSO6104 oscilloscope. This feature works with any oscilloscope using an IVI-C driver. You can follow the basic steps using your particular scope. For use with a Tektronix scope, see Read Waveforms Using a Tektronix Scope.

1 Ensure all necessary software is installed. See “Quick-Control Oscilloscope Requirements” on page 14-21 for the list.

2 Ensure that your instrument is recognized by the VISA utility. In this case, open Keysight Connectivity Expert and make sure it recognizes the oscilloscope.

3 Create an instance of the oscilloscope.

```
% Instantiate an instance of the scope.
myScope = oscilloscope()
```

4 Discover available resources. A resource string is an identifier to the instrument. You must set it before connecting to the instrument.

```
% Find resources.
availableResources = resources(myScope)
```

This returns a resource string or an array of resource strings.

```
availableResources =
  TCPIP0::a-m6104a-004598.dhcp.mathworks.com::inst0::INSTR
```

5 Connect to the scope.

If multiple resources are available, use the VISA utility to verify the correct resource and set it.

```
myScope.Resource = 'TCPIP0::a-m6104a-004598::inst0::INSTR';

% Connect to the scope.
connect(myScope);
```

6 Configure the oscilloscope.

You can configure any of the scope’s properties that are able to be set. In this example enable channel 1 and configure various acquisition settings as shown.

```
% Automatically configure the scope based on the input signal.
autoSetup(myScope);

% Set the acquisition time to 0.01 second.
myScope.AcquisitionTime = 0.01;

% Set the acquisition to collect 2000 data points.
myScope.WaveformLength = 2000;

% Set the trigger mode to normal.
myScope.TriggerMode = 'normal';

% Set the trigger level to 0.1 volt.
myScope.TriggerLevel = 0.1;

% Enable channel 1.
enableChannel(myScope, 'CH1');

% Set the vertical coupling to AC.
setVerticalCoupling (myScope, 'CH1', 'AC');
```

```
% Set the vertical range to 5.0.
setVerticalRange (myScope, 'CH1', 5.0);
```

7. Communicate with the instrument. For example, read a waveform.

   In this example, the `readWaveform` function returns the waveform that was acquired using the front panel of the scope. The function can also initiate an acquisition on the enabled channel and then return the waveform after the acquisition. For examples on all the use cases for this function, see `getWaveform`.

   ```
   % Acquire the waveform.
   waveformArray = readWaveform(myScope);

   % Plot the waveform and assign labels for the plot.
   plot(waveformArray);
   xlabel('Samples');
   ylabel('Voltage');
   ```

8. After configuring the instrument and retrieving its data, close the session and remove it from the workspace.

   ```
   disconnect(myScope);
   clear myScope;
   ```

For a list of supported functions for use with Quick-Control Oscilloscope, see "Quick-Control Oscilloscope Functions" on page 14-26.

See Also

Related Examples

- "Read a Waveform Using a Tektronix Scope" on page 14-24
- "Quick-Control Oscilloscope Functions" on page 14-26
- "Quick-Control Oscilloscope Properties" on page 14-28

Read a Waveform Using a Tektronix Scope

Reading a waveform with a Tektronix scope using Quick-Control Oscilloscope is basically the same workflow as described in the Read Waveforms Using Quick-Control Oscilloscope example using a Keysight scope with VISA. But the resource and driver information is different.

If you use the `resources` function, instead of getting a VISA resource string as shown in step 4 of the previous example, you will get the interface resource of the Tektronix scope. For example:

```
% Find resources.
availableResources = resources(myScope)
```

This returns the interface resource information.

```
availableResources =
  GPIB0::AGILENT::7::10
```

Where `gpib` is the interface being used, `keysight` is the interface type for the adaptor that the Tektronix scope is connected to, and the numbers are interface constructor parameters.

If you use the `drivers` function, you get information about the driver and its supported instrument models. For example:

```
% Get driver information.
driverlist = drivers(myScope)
```

This returns the driver and instrument model information.

```
Driver: tekronix
Supported Models:
  TDS200, TDS1000, TDS2000, TDS1000B, TDS2000B, TPS2000
  TDS3000, TDS3000B, MSO4000, DPO4000, DPO7000, DPO7000B
```

This example shows the general workflow to use for the Quick-Control Oscilloscope for a Tektronix scope. This feature works with any supported oscilloscope model. You can follow the basic steps using your particular scope.

1. Create an instance of the oscilloscope.

   ```
   % Instantiate an instance of the scope.
   myScope = oscilloscope()
   ```
2. Discover available resources. A resource string is an identifier to the instrument. You must set it before connecting to the instrument.

   ```
   % Find resources.
   availableResources = resources(myScope)
   ```

   This returns a resource string or an array of resource strings.

   ```
   availableResources =
     GPIB0::AGILENT::7::10
   ```

   Where `gpib` is the interface being used, `keysight` is the interface type for the adaptor that the Tektronix scope is connected to, and the numbers are interface constructor parameters.
3. Connect to the scope.

```
% Connect to the scope.
connect(myScope);
```

4 Configure the oscilloscope.

You can configure any of the scope’s properties that are able to be set. In this example enable channel 1 and set acquisition time as shown. You can see examples of other acquisition parameters in step 6 of the previous example.

```
% Set the acquisition time to 0.01 second.
myScope.AcquisitionTime = 0.01;

% Set the acquisition to collect 2000 data points.
set(myScope, 'WaveformLength', 2000);

% Enable channel 1.
enableChannel(myScope, 'CH1');
```

5 Communicate with the instrument. For example, read a waveform.

In this example, the `readWaveform` function returns the waveform that was acquired using the front panel of the scope. The function can also initiate an acquisition on the enabled channel and then return the waveform after the acquisition. For examples on all the use cases for this function, see `getWaveform`.

```
% Acquire the waveform.
waveformArray = readWaveform(myScope);

% Plot the waveform and assign labels for the plot.
plot(waveformArray);
xlabel('Samples');
ylabel('Voltage');
```

6 After configuring the instrument and retrieving its data, close the session and remove it from the workspace.

```
disconnect(myScope);
clear myScope;
```

For a list of supported functions for use with Quick-Control Oscilloscope, see “Quick-Control Oscilloscope Functions” on page 14-26.

See Also

Related Examples

- “Read Waveforms Using the Quick-Control Oscilloscope” on page 14-22
- “Quick-Control Oscilloscope Functions” on page 14-26
- “Quick-Control Oscilloscope Properties” on page 14-28

Quick-Control Oscilloscope Functions

The `oscilloscope` function can use the following special functions, in addition to standard functions such as `connect` and `disconnect`.

Function	Description
`autoSetup`	Automatically configures the instrument based on the input signal. `autoSetup(myScope);`

`disableChannel`	Disables the oscilloscope's channel(s).
	
	`disableChannel(myScope, 'Channel1');`
	`disableChannel(myScope, {'Channel1', 'Channel2'});`

`enableChannel`	Enables the oscilloscope's channel(s) from which waveform(s) will be retrieved.
	
	`enableChannel(myScope, 'Channel1');`
	`enableChannel(myScope, {'Channel1', 'Channel2'});`

`drivers`	Returns a list of available drivers with their supported instrument models. `driverlist = drivers(myScope);`
`resources`	Retrieves a list of available resources of instruments. It returns a list of available VISA resource strings when using an IVI-C scope. It returns the interface resource information when using a Tektronix scope. `res = resources(myScope);`

`configureChannel`	Returns or sets specified oscilloscope control on selected channel. Possible controls are:
	
	• `'VerticalCoupling'`
	• `'VerticalOffset'`
	• `'VeticalRange'`
	• `'ProbeAttenuation'`
	
	`value = configureChannel(myScope,'Channel1','VerticalOffset');`
	`configureChannel(myScope,'Channel1','VerticalCoupling','AC');`

`getVerticalCoupling`	Returns the value of how the oscilloscope couples the input signal for the selected channel name as a MATLAB character vector. Possible values returned are `'AC'`, `'DC'`, and `'GND'`. `VC = getVerticalCoupling (myScope, 'Channel1');`
`getVerticalOffset`	Returns the location of the center of the range for the selected channel name as a MATLAB character vector. The units are volts. `VO = getVerticalOffset (myScope, 'Channel1');`
`getVerticalRange`	Returns the absolute value of the input range the oscilloscope can acquire for selected channel name as a MATLAB character vector. The units are volts. `VR = getVerticalRange (myScope, 'Channel1');`

Function	Description
`readWaveform`	Returns the waveform(s) displayed on the scope screen. Retrieves the waveform(s) from enabled channel(s). `w = readWaveform(myScope);`
`reset`	Resets the device to factory default state. `reset(myScope);`
`setVerticalCoupling`	Specifies how the oscilloscope couples the input signal for the selected channel name as a MATLAB character vector. Values are `'AC'`, `'DC'`, and `'GND'`. `setVerticalCoupling (myScope, 'Channel1', 'AC');`
`setVerticalOffset`	Specifies the location of the center of the range for the selected channel name as a MATLAB character vector. For example, to acquire a sine wave that spans from 0.0 to 10.0 volts, set this attribute to 5.0 volts. `setVerticalOffset (myScope, 'Channel1', 5);`
`setVerticalRange`	Specifies the absolute value of the input range the oscilloscope can acquire for the selected channel name as a MATLAB character vector. The units are volts. `setVerticalRange (myScope, 'Channel1', 10);`

Quick-Control Oscilloscope Properties

The Quick-Control Oscilloscope `oscilloscope` function can use the following properties.

Property	Description
`ChannelNames`	Read-only property that provides available channel names in a cell array.
`ChannelsEnabled`	Read-only property that provides currently enabled channel names in a cell array.
`Status`	Read-only property that indicates the communication status. Valid values are `open` or `closed`.
`Timeout`	Use to get or set a timeout value. Value cannot be negative number. Default is `10` seconds.
`AcquisitionTime`	Use to get or set acquisition time value. Used to control the time in seconds that corresponds to the record length. Value must be a positive, finite number.
`AcquisitionStartDelay`	Use to set or get the length of time in seconds from the trigger event to first point in waveform record. If positive, the first point in the waveform occurs after the trigger. If negative, the first point in the waveform occurs before the trigger.
`TriggerMode`	Use to set the triggering behavior. Values are: `'normal'` – the oscilloscope waits until the trigger the user specifies occurs. `'auto'` – the oscilloscope automatically triggers if the configured trigger does not occur within the oscilloscope’s timeout period.
`TriggerSlope`	Use to set or get trigger slope value. Valid values are `falling` or `rising`.
`TriggerLevel`	Specifies the voltage threshold in volts for the trigger control.
`TriggerSource`	Specifies the source the oscilloscope monitors for a trigger. It can be channel name or other values.
`Resource`	Set up before connecting to instrument. Set with value of your instrument’s resource string, for example: `set(myScope, 'Resource', 'TCPIP0::a-m6104a-004598::inst0::INSTR');`

Property	Description
DriverDetectionMode	Optionally used to set up criteria for connection. Valid values are auto or manual. Default is auto. auto means you do not have to set a driver name before connecting to an instrument. If set to manual, a driver name must be provided before connecting.
Driver	Use only if set DriverDetectionMode to manual. Then use to give driver name. Only use if driver name cannot be figured out programmatically.

Quick-Control Function Generator Requirements

You can use the Quick-Control Function Generator for any function generator that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. This easy-to-use function generator, or fgen, is used for simplified fgen control and waveform generation.

Create the Quick-Control Function Generator object using the Instrument Control Toolbox `fgen` function. It simplifies controlling function generators and performs arbitrary waveform generations without dealing with the underlying drivers.

You can use the Quick-Control Function Generator for any function generator that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. This fgen object is easy to use.

The documentation examples use a specific instrument, a Tektronix AFG 3022B function generator. This feature works with any instrument that has IVI-C fgen class drivers. You can follow the basic steps, using your particular instrument.

To use the Quick-Control Function Generator for an IVI-C fgen, ensure the following software is installed. Most components are installed by the Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces. To install the support package, see “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11.

- Windows 64-bit platforms
- VISA shared components (installed by the support package)
- VISA (installed by the support package)

  Note, the examples use Keysight VISA, but you can use any vendor’s implementation of VISA.
- National Instruments IVI compliance package NICP 4.1 or later (installed by the support package)
- Your instrument’s device-specific driver. If you do not already have it, go to your instrument vendor's website and download the IVI-C driver for your specific instrument.

You can use `instrhwinfo` to confirm that the required software is installed.

```
% Check that the software is properly installed.
instrhwinfo('ivi')
```

See Also

Related Examples

- “Generate Standard Waveforms Using the Quick-Control Function Generator” on page 14-31
- “Generate Arbitrary Waveforms Using Quick-Control Function Generator” on page 14-33
- “Quick-Control Function Generator Functions” on page 14-35
- “Quick-Control Function Generator Properties” on page 14-37

Generate Standard Waveforms Using the Quick-Control Function Generator

This example shows how to use the Quick-Control Function Generator to generate a standard waveform. To generate an arbitrary waveform, see Generate Arbitrary Waveforms Using Quick-Control Function Generator. Quick-Control Function Generator works with any function generator using an IVI-C driver as long as the instrument and the driver support the functionality. You can follow the basic steps using your particular function generator. This example uses Keysight VISA, but you can use any vendor's implementation of VISA.

In this example, an electronic test engineer wants to create a simple sine waveform to test the clock operating range of a digital circuit.

1. Ensure all necessary software is installed. See “Quick-Control Function Generator Requirements” on page 14-30 for the list.
2. Create an instance of the function generator.

   ```
   % Instantiate an instance of the fgen.
   myFGen = fgen();
   ```
3. Discover available resources. A resource string is an identifier to the instrument. You must set it before connecting to the instrument.

   ```
   % Find resources.
   availableResources = resources(myFGen)
   ```

   This returns a resource string or an array of resource strings, for example:

   ```
   ans =

   ASRL::COM1
   GPIB0::INTFC
   GPIB0::10::INSTR
   PXI0::MEMACC
   TCPIP0::172.28.16.153::inst0::INSTR
   TCPIP0::172.28.16.174::inst0::INSTR
   ```
4. Set the resource for the function generator you want to communicate with.

   ```
   myFGen.Resource = 'GPIB0::10::INSTR';
   ```
5. Connect to the function generator.

   ```
   connect(myFGen);
   ```
6. Specify the channel name from which the function generator produces the waveform.

   ```
   selectChannel(myFGen, '1');
   ```
7. Configure the function generator.

   You can configure any of the instrument’s properties that are settable. Configure the waveform to be a continuous sine wave and then configure various settings as shown.

   ```
   % Set the type of waveform to a sine wave.
   myFGen.Waveform = 'sine';

   % Set the output mode to continuous.
   myFGen.Mode = 'continuous';
   ```

```
% Set the load impedance to 50 Ohms.
myFGen.OutputImpedance = 50;

% Set the frequency to 2500 Hz.
myFGen.Frequency = 2500;

% Set the amplitude to 1.2 volts.
myFGen.Amplitude = 1.2;

% Set the offset to 0.4 volts.
myFGen.Offset = 0.4;
```

8. Enable signal generation with the instrument, for example, output signals.

   In this example, the `enableOutput` function enables the function generator to produce a signal that appears at the output connector.

   ```
   % Enable the output of signals.
   enableOutput(myFGen);
   ```

   When you are done, disable the output.

   ```
   % Disable the output of signals.
   disableOutput(myFGen);
   ```

9. After configuring the instrument and generating a signal, close the session and remove it from the workspace.

   ```
   disconnect(myFGen);
   clear myFgen;
   ```

For a list of supported functions for use with Quick-Control Function Generator, see “Quick-Control Function Generator Functions” on page 14-35.

See Also

Related Examples

- “Generate Arbitrary Waveforms Using Quick-Control Function Generator” on page 14-33
- “Quick-Control Function Generator Functions” on page 14-35
- “Quick-Control Function Generator Properties” on page 14-37

Generate Arbitrary Waveforms Using Quick-Control Function Generator

This example shows how to use Quick-Control Function Generator to generate an arbitrary waveform. To generate a standard waveform, see Generate Standard Waveforms Using Quick-Control Function Generator. Quick-Control Function Generator works with any function generator using an IVI-C driver as long as the instrument and the driver support the functionality. You can follow the basic steps using your particular function generator. This example uses Keysight VISA, but you can use any vendor's implementation of VISA.

In this example, an electronic design engineer wants to generate a complex waveform with MATLAB, then download them into the function/arbitrary waveform generator and output them one after the other, and then finally remove the downloaded waveforms afterward. In this example we are using the GPIB interface.

1. Ensure all necessary software is installed. See “Quick-Control Function Generator Requirements” on page 14-30 for the list.
2. Create an instance of the function generator.

   ```
   % Instantiate an instance of the fgen.
   myFGen = fgen();
   ```
3. Set the resource.

   ```
   myFGen.Resource = 'GPIB0::10::INSTR';
   ```
4. Connect to the function generator.

   ```
   connect(myFGen);
   ```
5. Specify the channel name from which the function generator produces the waveform.

   ```
   selectChannel(myFGen, '1');
   ```
6. Configure the function generator.

   You can configure any of the instrument’s properties that are settable. Configure the waveform to be a continuous arbitrary wave.

   ```
   % Set the type of waveform to an arbitrary wave.
   myFGen.Waveform = 'arb';

   % Set the output mode to continuous.
   myFGen.Mode = 'continuous';
   ```
7. Communicate with the instrument.

   In this example, create the waveform, then download it to the function generator using the `downloadWaveform` function. Then enable the output using the `enableOutput` function, and then remove the waveform using the `removeWaveform` function.

   ```
   % Create the waveform.
   w1 = 1:0.001:2;

   % Download the waveform to the function generator.
   h1 = downloadWaveform (myFGen, w1);

   % Enable the output.
   enableOutput(myFGen);
   ```

When you are done, remove the waveforms.

```
% Remove the waveform.
removeWaveform(myFGen);
```

8 After communicating with the instrument, close the session and remove it from the workspace.

```
disconnect(myFGen);
clear myFgen;
```

For a list of supported functions for use with Quick-Control Function Generator, see “Quick-Control Function Generator Functions” on page 14-35.

See Also

Related Examples

- “Generate Standard Waveforms Using the Quick-Control Function Generator” on page 14-31
- “Quick-Control Function Generator Functions” on page 14-35
- “Quick-Control Function Generator Properties” on page 14-37

Quick-Control Function Generator Functions

The `fgen` function uses the following functions, in addition to standard functions such as `connect` and `disconnect`.

Function	Description
`selectChannel`	Specifies the channel name from which the function generator produces the waveform. Example: `selectChannel(myFGen, '1');`
`drivers`	Returns a list of available function generator instrument drivers. Example: `driverlist = drivers(myFGen);` See the note following this table about using a SCPI-based driver for Keysight function generators.
`resources`	Retrieves a list of available instrument resources. It returns a list of available VISA resource strings when using an IVI-C function generator. Example: `res = resources(myFGen);`
`selectWaveform`	Specifies which arbitrary waveform the function generator produces. Example: `selectWaveform (myFGen, wh);` where `wh` is the waveform handle you are selecting.

`downloadWaveform`	Downloads an arbitrary waveform to the function generator. If you provide an output variable, a waveform handle is returned. It can be used in the `selectWaveform` and `removeWaveform` functions.
	
	If you don't provide an output variable, function generator overwrites the waveform when a new waveform is downloaded and deletes it upon disconnection.
	
	Example:
	
	`% Download the following waveform to fgen`
	`w = 1:0.001:2;`
	`downloadWaveform (myFGen, w);`
	
	`% Download a waveform to fgen and return`
	`   a waveform handle`
	`wh = downloadWaveform (myFGen, w);`

Function	Description

`removeWaveform`	Removes a previously created arbitrary waveform from the function generator's memory. If a waveform handle is provided, it removes the waveform represented by the waveform handle.
	
	Example:
	
	`% Remove a waveform from fgen with waveform handle 10000`
	`removeWaveform (myFGen, 10000);`

`enableOutput`	Enables the function generator to produce a signal that appears at the output connector. This function produces a waveform defined by the `Waveform` property. If the `Waveform` property is set to `'Arb'`, the function uses the latest internal waveform handle to output the waveform. `enableOutput (myFGen);`
`disableOutput`	Disables the signal that appears at the output connector. Disables the selected channel. `disableOutput (myFGen);`
`reset`	Sets the function generator to factory default state.

Using a SCPI-based Driver for Keysight Function Generators

If you are using a SCPI-based Keysight function generator such as the 33220A, you will see the following when you use the `drivers` function on an fgen object `myFGen`.

```
driverlist = drivers(myFGen);

driverlist =

Driver: Agilent332x0_SCPI
Supported Models:
   33210A, 33220A, 33250A
```

The `_SCPI` after the instrument name indicates this is using a SCPI driver instead of the IVI driver.

Using Properties

For a list of supported properties for use with Quick-Control Function Generator, see Quick-Control Function Generator Properties.

Quick-Control Function Generator Properties

The fgen function can use the following properties.

Property	Description
AMDepth	Specifies the extent of Amplitude modulation the function generator applies to the carrier signal. The units are a percentage of full modulation. At 0% depth, the output amplitude equals the carrier signal's amplitude. At 100% depth, the output amplitude equals twice the carrier signal's amplitude. This property affects function generator behavior only when the Mode is set to 'AM' and ModulationResource is set to 'internal'.
Amplitude	Specifies the amplitude of the standard waveform. The value is the amplitude at the output terminal. The units are volts peak-to-peak (Vpp). For example, to produce a waveform ranging from -5.0 to +5.0 volts, set this value to 10.0 volts. Does not apply if Waveform is of type 'Arb'.
ArbWaveformGain	Specifies the factor by which the function generator scales the arbitrary waveform data. Use this property to scale the arbitrary waveform to ranges other than -1.0 to +1.0. When set to 2.0, the output signal ranges from -2.0 to +2.0 volts. Only applies if Waveform is of type 'Arb'.
BurstCount	Specifies the number of waveform cycles that the function generator produces after it receives a trigger. Only applies if Mode is set to 'burst'.
ChannelNames	This read-only property provides available channel names in a cell array.
Driver	This property is optional. Use only if necessary to specify the underlying driver used to communicate with an instrument. If the DriverDetectionMode property is set to 'manual', use the Driver property to specify the instrument driver.
DriverDetectionMode	Sets up criteria for connection. Valid values are 'auto' and 'manual'. The default value is 'auto', which means you do not need to set a driver name before connecting to an instrument. If set to 'manual', a driver name needs to be provided using the Driver property before connecting to instrument.
FMDeviation	Specifies the maximum frequency deviation the modulating waveform applies to the carrier waveform. This deviation corresponds to the maximum amplitude level of the modulating signal. The units are Hertz (Hz). This property affects function generator behavior only when Mode is set to 'FM' and ModulationSource is set to 'internal'.
Frequency	Specifies the rate at which the function generator outputs an entire arbitrary waveform when Waveform is set to 'Arb'. It specifies the frequency of the standard waveform when Waveform is set to standard waveform types. The units are Hertz (Hz).

Property	Description
`Mode`	Specifies run mode. Valid values are `'continuous'`, `'burst'`, `'AM'`, or `'FM'`. Specifies how the function generator produces waveforms. It configures the instrument to generate output continuously or to generate a discrete number of waveform cycles based on a trigger event. It can also be set to AM and FM.
`ModulationFrequency`	Specifies the frequency of the standard waveform that the function generator uses to modulate the output signal. The units are Hertz (Hz). This attribute affects function generator behavior only when `Mode` is set to `'AM'` or `'FM'` and the `ModulationSource` attribute is set to `'internal'`.
`ModulationSource`	Specifies the signal that the function generator uses to modulate the output signal. Valid values are `'internal'` and `'external'`. This attribute affects function generator behavior only when `Mode` is set to `'AM'` or `'FM'`.
`ModulationWaveform`	Specifies the standard waveform type that the function generator uses to modulate the output signal. This affects function generator behavior only when `Mode` is set to `'AM'` or `'FM'` and the `ModulationSource` is set to `'internal'`. Valid values are `'sine'`, `'square'`, `'triangle'`, `'RampUp'`, `'RampDown'`, and `'DC'`.
`Offset`	Uses the standard waveform DC offset as input arguments if the waveform is not of type `'Arb'`. Use Arb Waveform Offset as input arguments if the waveform is of type `'Arb'`. Specifies the DC offset of the standard waveform when Waveform is set to standard waveform. For example, a standard waveform ranging from +5.0 volts to 0.0 volts has a DC offset of 2.5 volts. When Waveform is set to `'Arb'`, this property shifts the arbitrary waveform's range. For example, when it is set to 1.0, the output signal ranges from 2.0 volts to 0.0 volts.
`OutputImpedance`	Specifies the function generator's output impedance at the output connector.
`Resource`	Set this before connecting to the instrument. It is the VISA resource string for your instrument.
`SelectedChannel`	Returns the selected channel name that was set using the `selectChannel` function.
`StartPhase`	Specifies the horizontal offset in degrees of the standard waveform the function generator produces. The units are degrees of one waveform cycle. For example, a 180-degree phase offset means output generation begins halfway through the waveform.
`Status`	This read-only property indicates the communication status of your instrument session. It is either `'open'` or `'closed'`.
`TriggerRate`	Specifies the rate at which the function generator's internal trigger source produces a trigger, in triggers per second. This property affects function generator behavior only when the `TriggerSource` is set to `'internal'`. Only applies if `Mode` is set to `'burst'`.

Property	Description
`TriggerSource`	Specifies the trigger source. After the function generator receives a trigger, it generates an output signal if `Mode` is set to `'burst'`. Valid values are `'internal'` or `'external'`.
`Waveform`	Uses the waveform type as an input argument. Valid values are `'Arb'`, for an arbitrary waveform, or these standard waveform types – `'Sine'`, `'Square'`, `'Triangle'`, `'RampUp'`, `'RampDown'`, and `'DC'`.

Quick-Control RF Signal Generator Requirements

You can use the Quick-Control RF Signal Generator for any RF signal generator that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver.

Create the Quick-Control RF Signal Generator object using the Instrument Control Toolbox `rfsiggen` function. It simplifies controlling RF signal generators and performs waveform generations. This feature works with any instrument that has IVI-C rfsiggen class drivers.

To use the Quick-Control RF Signal Generator for an IVI-C RF signal generator, ensure the following software is installed. Most components are installed by the Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces, but you can also install them separately. To install the support package, see “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11.

- Windows 64-bit platforms
- VISA shared components (installed by the support package)
- VISA (installed by the support package)

  Note, the examples use Keysight VISA, but you can use any vendor’s implementation of VISA.
- National Instruments IVI compliance package NICP 4.1 or later (installed by the support package)
- The device-specific driver for your instrument. If you do not already have it, go to your instrument vendor's website and download the IVI-C driver for your specific instrument.

You can use `instrhwinfo` to confirm that the required software is installed.

```
% Check that the software is properly installed.
instrhwinfo('ivi')
```

See Also

More About

- “Quick-Control RF Signal Generator Functions” on page 14-41
- “Quick-Control RF Signal Generator Properties” on page 14-43
- “Download and Generate Signals with RF Signal Generator” on page 14-45

Quick-Control RF Signal Generator Functions

The `rfsiggen` function uses the following functions, in addition to standard Instrument Control Toolbox functions such as `connect` and `disconnect`.

Function	Description
`drivers`	Return a list of available RF signal generator instrument drivers with their supported instrument models. Example: `driverlist = drivers(myRFSigGen);` where my `myRFSigGen` is the name of the `rfsiggen` object.
`resources`	Retrieve a list of available instrument resources. It returns a list of available VISA resource strings when using an IVI-C RF signal generator. Example: `res = resources(myRFSigGen);` where my `myRFSigGen` is the name of the `rfsiggen` object.

`download`	Download an arbitrary waveform to the RF signal generator. It accepts a complex vector of doubles containing the `IQData` and a double defining the `SampleRate` of the signal.
	
	Example:
	
	`rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRfSigGen')`
	`    IQData = (-0.98:0.02:1) + 1i*(-0.98:0.02:1);`
	`    SampleRate = 800000;`
	`    download(rf, IQData, SampleRate)`

`start`	Enable the RF signal generator signal output and modulation output. It takes a double value for each of the three required arguments: `CenterFrequency` specified in Hz, `OutputPower` specified in dBm, and `LoopCount`, which represents the number of times the waveform should be repeated.
	
	Example:
	
	`rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRfSigGen')`
	`    CenterFrequency = 4000000`
	`    OutputPower = 0`
	`    LoopCount = inf`
	`    start(rf, CenterFrequency, OutputPower, LoopCount)`

`stop`	Stop the RF signal generator signal output and modulation output. `stop(rf);` where `rf` is the name of the `rfsiggen` object.
`reset`	Set the RF signal generator to factory default state.

See Also

More About

- "Quick-Control RF Signal Generator Requirements" on page 14-40
- "Quick-Control RF Signal Generator Properties" on page 14-43
- "Download and Generate Signals with RF Signal Generator" on page 14-45

Quick-Control RF Signal Generator Properties

The Quick-Control RF Signal Generator can use the following properties on the `rfsiggen`, `download`, or `start` functions. See the examples to learn how to set the properties.

Property	Description
`CenterFrequency`	Used on the `start` function, this argument is the center frequency for the waveform, specified as a double in Hz.
`Driver`	Used on the `rfsiggen` function, this argument specifies the underlying driver used to communicate with an instrument as a string. It is optional, and, if not specified, the driver is auto-detected.
`IQData`	Used on the `download` function, this argument specifies the IQ data to use in the download.
`LoopCount`	Used on the `start` function, this argument is the number of times the waveform should be repeated, specified as a double.
`OutputPower`	Used on the `start` function, this argument is the output power, specified as a double in dBm.
`Resource`	The VISA resource string for your instrument, specified as a string. Set this before connecting to the instrument. It is optional during object creation, and can be used if you know the resource string for your instrument. Otherwise you can set it after object creation.
`SampleRate`	Used on the `download` function, this argument specifies the sample rate to use in the download.

Set Driver or Resource During Object Creation

You can optionally set the Driver and Resource property values during the `rfsiggen` object creation.

The `Driver` property specifies the underlying driver used to communicate with an instrument, and is specified as a string. It is optional, and if not specified the driver is auto-detected.

The `Resource` property specifies the VISA resource string for your instrument, and is specified as a string. It is optional and can be used if you know the resource string for your instrument.

This example shows how to create the RF Signal Generator object `rf` and specify the resource string shown and a driver named `AgRFSigGen`.

```
rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRFSigGen')
```

Set IQ Data and Sample Rate for Download

You can set the `IQData` and `SampleRate` property values during the download operation.

This example shows how to create the RF Signal Generator object, assign values to the properties, and then perform the download.

```
rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRFSigGen')
IQData = (-0.98:0.02:1) + 1i*(-0.98:0.02:1);
SampleRate = 500000;
download(rf, IQData, SampleRate)
```

Set Signal Generation Properties

You can set property values that are used when you start the RF signal generator signal output and modulation output with the `start` function.

This example shows how to create the RF Signal Generator object, assign values to the properties, and then perform the signal generation.

```
rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgilentRFSigGen')
CenterFrequency = 2000000
OutputPower = 0
LoopCount = inf
start(rf, CenterFrequency, OutputPower, LoopCount)
```

See Also

More About

- “Quick-Control RF Signal Generator Requirements” on page 14-40
- “Quick-Control RF Signal Generator Functions” on page 14-41
- “Download and Generate Signals with RF Signal Generator” on page 14-45

Download and Generate Signals with RF Signal Generator

In this section...
“Create an RF Signal Generator Object” on page 14-45
“Download a Waveform” on page 14-46
“Generate Signal and Modulation Output” on page 14-47

Create an RF Signal Generator Object

You create an `rfsiggen` object to communicate with RF signal generators. You must specify a resource, either when you create the object or after object creation. The `Resource` property is the VISA resource string for the instrument.

You can optionally specify a driver either during or after object creation using the `Driver` property. If you don't specify one it is auto-detected.

Create an RF Signal Generator Object and Set Resource and Driver

You can create the `rfsiggen` object and set the `Resource` and `Driver` during object creation. If those properties are valid, it automatically connects to the instrument.

This syntax shows how to create the RF Signal Generator object and connect using the specified resource string and driver.

```
rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRfSigGen')
```

Create an RF Signal Generator Object without Setting Resource and Driver

You can create the `rfsiggen` object without setting the Resource or Driver, and then set it after object creation.

1. Create the RF Signal Generator object with no arguments.

   ```
   rf = rfsiggen;
   ```
2. Find available resources using the `resources` function.

   ```
   ResourceList = resources(rf)

   ResourceList =

      3x1 cell array

      {'ASRL::COM1'}
      {'ASRL::COM3'}
      'TCPIP0::172.28.22.99::inst0::INSTR'
   ```

   In this case, it finds two COM ports that could host an instrument, and the VISA resource string of an RF signal generator.
3. Set the RF Signal Generator resource using the `Resource` property, which is the VISA resource string.

   ```
   rf.Resource = 'TCPIP0::172.28.22.99::inst0::INSTR';
   ```

4 List the drivers using the `drivers` function.

```
drivers(rf)

ans =

     Driver: AgRfSigGen_SCPI
     Supported Models:
     E4428C, E4438C

     Driver: RsRfSigGen_SCPI
     Supported Models:
     SMW200A, SMBV100A, SMU200A, SMJ100A, AMU200A, SMATE200A

     Driver: AgRfSigGen
     Supported Models:
     E4428C,E4438C,N5181A,N5182A,N5183A,N5171B,N5181B,N5172B
     N5182B,N5173B,N5183B,E8241A,E8244A,E8251A,E8254A,E8247C
```

In this case, it finds the drivers for a Keysight (formerly Agilent) SCPI-based RF signal generator, a Rohde & Shwartz SCPI-based generator, and another Keysight generator. You can see that it lists the supported models of the driver in each case.

5 Set the RF Signal Generator driver using the `Driver` property.

```
rf.Driver = 'AgRfSigGen';
```

6 You can now connect to the instrument.

```
connect(rf);
```

Download a Waveform

You can download an arbitrary waveform to an RF signal generator using the `download` function and assign the `IQData` and `SampleRate` to use. The `IQData` is a complex vector of doubles containing the IQ data to use.

This example shows how to download a waveform to your `rfsiggen` object and assign the `IQData` and `SampleRate` to use.

1 Create an `rfsiggen` object to communicate with an RF signal generator, using the VISA resource string and driver associated with your own instrument.

```
rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRfSigGen')
```

When you designate the `Resource` and `Driver` properties during object creation, it automatically connects to the instrument.

2 Assign the `IQData` and `SampleRate` variables to use in the download.

```
IQData = (-0.98:0.02:1) + 1i*(-0.98:0.02:1);
SampleRate = 800000;
```

3 Perform the download.

```
download(rf, IQData, SampleRate)
```

Generate Signal and Modulation Output

You can use the `start` function on an RF signal generator object to start signal output and modulation output. It takes a double value for each of the three required arguments: `CenterFrequency` specified in Hz, `OutputPower` specified in dBm, and `LoopCount`, which represents the number of times the waveform should be repeated.

This example shows how to enable signal output and modulation output for the RF signal generator, and assign the required arguments.

1. Create an `rfsiggen` object to communicate with an RF signal generator, using the VISA resource string and driver associated with your own instrument.

   ```
   rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRfSigGen')
   ```

   When you designate the `Resource` and `Driver` properties during object creation, it automatically connects to the instrument.
2. Assign the `CenterFrequency`, `OutputPower`, and `LoopCount` variables to use in the signal generation.

   ```
   CenterFrequency = 4000000
   OutputPower = 0
   LoopCount = inf
   ```
3. Start the signal generation.

   ```
   start(rf, CenterFrequency, OutputPower, LoopCount)
   ```

See Also

More About

- “Quick-Control RF Signal Generator Requirements” on page 14-40
- “Quick-Control RF Signal Generator Functions” on page 14-41
- “Quick-Control RF Signal Generator Properties” on page 14-43

Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI

When using IVI-C or VXI*plug&play* drivers, executing your code will generate additional files in the folder specified by executing the following code at the MATLAB prompt:

```
fullfile(tempdir,'ICTDeploymentFiles',sprintf('R%s',version('-release')))
```

On all supported platforms, a file with the name `MATLABPrototypeFor<driverName>.m` is generated, where `<driverName>` the name of the IVI-C or VXI*plug&play* driver. With 64-bit MATLAB on Windows, a second file by the name `<driverName>_thunk_pcwin64.dll` is generated. When creating your deployed application or shared library, manually include these generated files. For more information on including additional files refer to the MATLAB Compiler documentation.

Instrument Support Packages

- “Install the Ocean Optics Spectrometers Support Package” on page 15-2
- “Install the NI-SCOPE Oscilloscopes Support Package” on page 15-4
- “Install the NI-FGEN Function Generators Support Package” on page 15-5
- “Install the NI-DCPower Power Supplies Support Package” on page 15-6
- “Install the NI-DMM Digital Multimeters Support Package” on page 15-7
- “Install the NI-845x I2C/SPI Interface Support Package” on page 15-8
- “Install the Total Phase Aardvark I2C/SPI Interface Support Package” on page 15-9
- “Install the NI-Switch Hardware Support Package” on page 15-10
- “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11
- “Install the Keysight IO Libraries and VISA Interface Support Package” on page 15-12

Install the Ocean Optics Spectrometers Support Package

You can use Instrument Control Toolbox to communicate with Ocean Optics USB spectrometers. You can acquire data from the spectrometer and control it. Ocean Optics manufactures a broad line of USB-powered spectrometers covering the visible, near IR, and UV portions of the spectrum. You can use these spectrometers from MATLAB on Windows and Macintosh platforms.

The Instrument Control Toolbox Support Package for Ocean Optics Spectrometers lets you use MATLAB for comprehensive control of any spectrometer that is supported by the Ocean Optics OmniDriver software (version 2.12 or higher). You can perform many tasks, including:

- Acquire a spectrum
- Set the integration time
- Enable dark current and nonlinear spectral corrections
- View all connected devices

For a list of supported devices, see https://www.mathworks.com/hardware-support/ocean-optics-spectrometers.html.

Note This support package is being removed in R2018b. The Instrument Control Toolbox still supports use of Ocean Optics hardware as described here. Required MATLAB files will be installed with the toolbox. Required Ocean Optics files, such as instrument drivers, need to be installed separately from Ocean Optics. You can still use Ocean Optics instruments with the toolbox. To do so, install any required instrument drivers from Ocean Optics, if you do not already have them installed. Required files from Ocean Optics are available from the MathWorks web site: https://www.mathworks.com/hardware-support/ocean-optics-spectrometers.html.

This feature is available through the Hardware Support Packages. Using this installation process, download and install the following file(s) on your host computer:

- MATLAB Instrument Driver for Ocean Optics support
- Ocean Optics OmniDriver version 2.2 driver files
- An example that shows how to take measurements with an Ocean Optics spectrometer

---

Note You can use this support package on a host computer running on 64-bit Windows or 64-bit macOS operating systems that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for Ocean Optics Spectrometers:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the NI-SCOPE Oscilloscopes Support Package

You can use Instrument Control Toolbox to communicate with NI-SCOPE oscilloscopes. You can acquire waveform data from the oscilloscope and control it.

This feature is available through the Instrument Control Toolbox Support Package for NI-SCOPE Oscilloscopes. Using this installation process, you download and install the following file(s) on your host computer:

- MATLAB Instrument Driver for NI-SCOPE support
- National Instruments driver file: NI-SCOPE driver version 3.9.7

Note You can use this support package only on a host computer running a version of 64-bit Windows that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-SCOPE Oscilloscopes:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the NI-FGEN Function Generators Support Package

You can use the Instrument Control Toolbox to communicate with NI-FGEN function generators. You can control and configure the function generator, and perform tasks such as generating sine waves.

This feature is available through the Instrument Control Toolbox Support Package for NI-FGEN Function Generators. Using this installation process, you download and install the following file(s) on your host computer:

- MATLAB Instrument Driver for NI-FGEN support
- National Instruments driver file: NI-FGEN driver version 2.9.1

Note You can use this support package only on a host computer running a version of 64-bit Windows that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-FGEN Function Generators:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the NI-DCPower Power Supplies Support Package

You can use Instrument Control Toolbox to communicate with NI-DCPower power supplies. You can control and take digital measurements from a power supply, such as the NI PXI 4011 triple-output programmable DC power supply.

This feature is available through the Instrument Control Toolbox Support Package for NI-DCPower Power Supplies. Using this installation process, you download and install the following file(s) on your host computer:

- MATLAB Instrument Driver for NI-DCPower support
- National Instruments NI-DCPower driver file
- Example that shows how to take digital measurements from an NI-DCPower power supply

Note You can use this support package only on a host computer running a version of 64-bit Windows that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-DCPower Power Supplies:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the NI-DMM Digital Multimeters Support Package

You can use Instrument Control Toolbox to communicate with NI-DMM digital multimeters. You can control and take measurements from a digital multimeter, such as measuring voltage or resistance.

This feature is available through the Instrument Control Toolbox Support Package for NI-DMM Digital Multimeters. Using this installation process, you download and install the following file(s) on your host computer:

- MATLAB Instrument Driver for NI-DMM support
- National Instruments NI-DMM driver file
- Example that shows how to take digital measurements from a NI-DMM digital multimeter

Note You can use this support package only on a host computer running a version of 64-bit Windows that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-DMM Digital Multimeters:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the NI-845x I2C/SPI Interface Support Package

For the Instrument Control Toolbox I2C and SPI interfaces, you can use either a Total Phase Aardvark host adaptor or an NI-845x adaptor. To use the I2C or SPI interface with the NI-845x adaptor, you must download this Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

To use the NI-845x driver, download and install the Instrument Control Toolbox Support Package for NI-845x I2C/SPI Interface, which includes the following files on your host computer:

- National Instruments NI-845x adaptor driver file
- Example that shows how to use the NI-854x adaptor with the I2C interface

Note You can use this support package only on a host computer running a version of 64-bit Windows that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-845x I2C/SPI Interface:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the Total Phase Aardvark I2C/SPI Interface Support Package

For the Instrument Control Toolbox I2C and SPI interfaces, you can use either a Total Phase Aardvark host adaptor or an NI-845x adaptor. To use the I2C or SPI interface with the Aardvark adaptor, you must download this Hardware Support Package to obtain the necessary files. You must also download the USB device driver from the vendor.

The Instrument Control Toolbox Support Package for Total Phase Aardvark I2C/SPI Interface downloads and installs the Total Phase Aardvark host adaptor driver file on your host computer. Examples of using the Aardvark adaptor with the I2C interface can be found in the Instrument Control Toolbox documentation. For more information on using Aardvark with the I2C and SPI interfaces, see “I2C Interface Overview” on page 9-2 and “SPI Interface Overview” on page 10-2.

Note For R2018b and R2018a, you cannot use the Aardvark adaptor for I2C or SPI interfaces on the macOS platform. You can still use it on Windows and Linux. For releases prior to R2018a, you can use it on all three platforms, including the Mac.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for Total Phase Aardvark I2C/SPI Interface:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the NI-Switch Hardware Support Package

You can use Instrument Control Toolbox to communicate with NI-Switch instruments. For example, you can control a relay box such as the NI PXI-2586 10-channel relay switch.

This feature is available through the Instrument Control Toolbox Support Package for NI-Switch Hardware. Using this installation process, you download and install the following file(s) on your host computer

- MATLAB Instrument Driver for NI-Switch support
- National Instruments NI-Switch driver file
- Example that shows how to control an NI-Switch relay switch

Note You can use this support package only on a host computer running a version of 64-bit Windows that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-Switch Hardware:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the National Instruments VISA and ICP Interfaces Support Package

The Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces lets you use the Quick Control Oscilloscope and Quick Control Function Generator interfaces.

After you download and install the support package, you can use the Quick Control interfaces to communicate with oscilloscopes and function generators.

The support package installs the following files on your host computer:

- MATLAB Instrument Driver for Quick Control Oscilloscope and Quick Control Function Generator
- VISA shared components
- VISA
- National Instruments IVI compliance package NICP 4.1 or later
- The Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces documentation

---

Note You can use this support package only on a host computer running a version of 64-bit Windows that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

Install the Keysight IO Libraries and VISA Interface Support Package

The Instrument Control Toolbox Support Package for Keysight IO Libraries and VISA Interface simplifies the use of Keysight (formerly Agilent) VISA by installing the necessary software components, such as the IO libraries and VISA shared components.

After you download and install the support package, you can use the VISA interface to communicate with Keysight instruments.

The support package installs the following files on your host computer:

- MATLAB Instrument Driver for Keysight VISA
- Keysight IO libraries
- Keysight VISA shared components

Installing the Support Package

To install the Instrument Control Toolbox Support Package for Keysight IO Libraries and VISA Interface:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.

In the Add-On Explorer, scroll to the **Hardware Support Packages** section, and click **show all** to find your support package.

To uninstall support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.

To update existing support packages:

On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Check for Updates > Hardware Support Packages**.

For more information about using Add-On Explorer, see “Get and Manage Add-Ons”.

16

Using Generic Instrument Drivers

This chapter describes the use of generic drivers for controlling instruments from the MATLAB Command window, using the Instrument Control Toolbox software.

- “Generic Drivers: Overview” on page 16-2
- “Writing a Generic Driver” on page 16-3
- “Using Generic Driver with Test & Measurement Tool” on page 16-7
- “Using a Generic Driver at Command Line” on page 16-9

Generic Drivers: Overview

Generic drivers allow the Instrument Control Toolbox software to communicate with devices or software that do not use industry-standard drivers or protocols.

Typical cases, but not the only possibilities, are instruments that offer access through a COM interface (where the instrument can be accessed as an ActiveX® object from the MATLAB workspace), that use proprietary libraries, or that use custom MEX-files.

Because the generic nature of this feature does not lend itself to detailed discussion of specific instructions that work in all cases, the following sections of this chapter use an example to illustrate how to create and use a MATLAB generic instrument driver:

- “Writing a Generic Driver” on page 16-3
- “Using Generic Driver with Test & Measurement Tool” on page 16-7
- “Using a Generic Driver at Command Line” on page 16-9

Writing a Generic Driver

In this section...
"Creating the Driver and Defining Its Initialization Behavior" on page 16-3
"Defining Properties" on page 16-4
"Defining Functions" on page 16-6

Creating the Driver and Defining Its Initialization Behavior

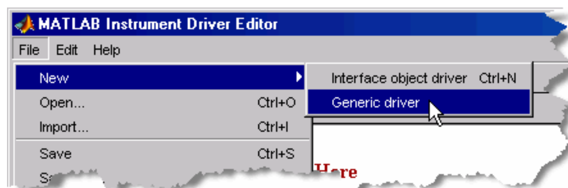
In this example, the generic "instrument" that you control is Microsoft Internet Explorer® (IE), which is represented by a COM object. (This example works only on Windows systems.) Working through the example, you write a simple MATLAB instrument generic driver that allows the Instrument Control Toolbox software to communicate with a COM object. Using both a graphical interface and command-line code, with your driver you create an IE browser window, control its size, and specify what Web page it displays. The principles demonstrated in this example can be applied when writing a generic driver for any kind of instrument.

In this section, you create a new driver and specify what happens when an object is created for this driver.

1 Open the MATLAB Instrument Driver Editor from the MATLAB Command Window.

```
midedit
```

2 To make it known that this driver is a generic driver, in the MATLAB Instrument Driver Editor, select **File** > **New** > **Generic driver**, as shown.



3 Select **File** > **Save as**.

Navigate to the directory where you want to save your driver, and give it any name you want. This example uses the name `ie_drv`. Remember where you have saved your driver.

4 Select the `Summary` node in the driver editor window. Set the fields of this pane with any values you want. This example uses the following settings:

Manufacturer	`Microsoft`
Supported models	`IE`
Instrument type	`Browser`
Driver version	`1.0`

5 Select the node `Initialization and Cleanup`.

6 Click the **Create** tab.

This is where you define the code to execute when this driver is used to create a device object. This example identifies the COM object for Internet Explorer, and assigns the handle to that object as the `Interface` property of the device object being created.

7 Add the following lines of code to the **Create** tab:

```
ie = actxserver('internetexplorer.application');
obj.Interface = ie);
```

8 Click the **Connect** tab.

This is where you define the code to execute when you connect your device object to your instrument or software.

9 Add the following lines of code to the **Connect** tab:

```
ie = get(obj, 'Interface');
ie.Visible = 1);
ie.FullScreen = 0);
```

The first line gets `ie` as a handle to the COM object, based on the assignment in the **Create** code. The two lines after that set the window visibility and size.

Defining Properties

Writing properties for generic drivers in the MATLAB Instrument Driver Editor is a matter of writing straight code.

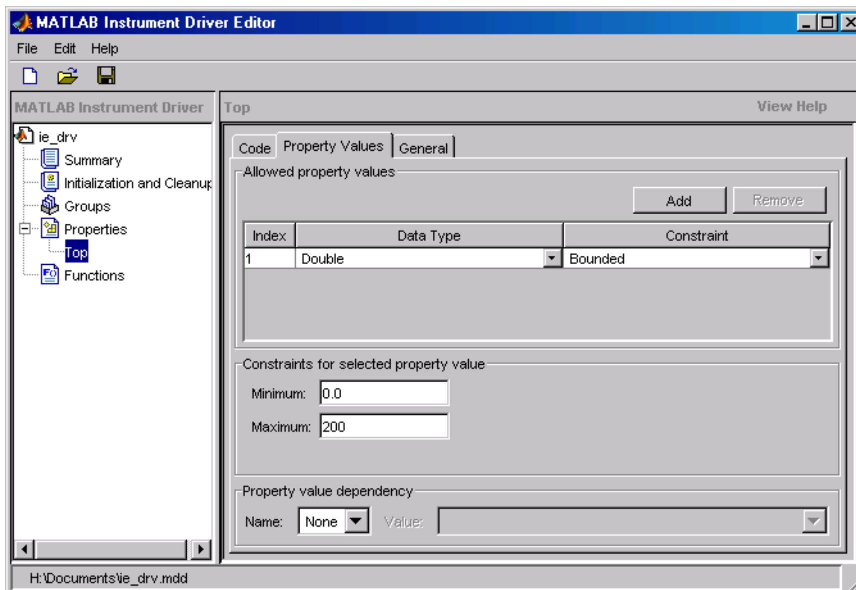
In this example, you define two properties. The first property uses the same name as the corresponding property of the COM object; the second property uses a different name from its corresponding COM object property.

Using the Same Name for a Property

The position of the IE browser window is determined by the `Top` and `Left` properties of its COM object. In the following steps, you make the `Top` property available to your device object through your generic driver. For this property, the name of the property is the same in both the COM object and in your device object.

1 Select the `Properties` node in the driver editor tree.
2 In the **Add property** field, enter the text `Top`, and click **Add**.
3 Expand the `Properties` node in the tree, and select the new node `Top`.
4 Click the **Property Values** tab. Your property can have a numeric value corresponding to screen pixels. For this example, you can limit the value of the property from `0` to `200`.
5 Make sure the **Data Type** field indicates `Double`. In the **Constraint** field, click the pull-down menu and select `Bounded`.
6 Keep the **Minimum** value of `0.0`, and enter a **Maximum** value of `200`.

Your driver editor window should look like the following figure.



Now that you have defined the data type and acceptable values of the property, you can write the code to be executed whenever the device object property is accessed by `get` or `set`.

7. Click the **Code** tab.

   The concept of reading the property is rather straightforward. When you get the `Top` property of the device object, the driver merely gets the value of the COM object's corresponding `Top` property. So all you need in the Get code function is to identify the COM object to get the information from.

8. Add the following code at the bottom of the function in the **Get code** pane:

```
ie = obj.Interface;
propertyValue = get(ie, propertyName);
```

   The first line gets `ie` as a handle to the COM object. Remember that the `Interface` property of the device object is set to this value back in the driver's **Create** code. The second line retrieves the value of the COM object's `Top` property, and assigns it to `propertyValue`, which is returned to the `get` function for the device object.

9. Add the following code at the bottom of the function in the **Set code** pane:

```
ie = get(obj, 'Interface');
ie.propertyName = propertyValue;
```

Using a Different Name for a Property

In the preceding steps, you created in your driver a device object property that has the same name as the property of the COM object representing your instrument. You can also create properties with names that do not match those of the COM object properties. In the following steps, you create a property called `Vsize` that corresponds to the IE COM object property `Height`.

1. Select the `Properties` node in the driver editor tree.
2. In the **Add property** field, enter the text `Vsize`, and click **Add**.
3. Expand the `Properties` node in the tree, and select the new node `Vsize`.
4. Click the **Property Values** tab. This property can have a numeric value corresponding to screen pixels, whose range you define as `200` to `800`.

5. Make sure the **Data Type** field indicates `Double`. In the **Constraint** field, click the pull-down menu and select `Bounded`.
6. Enter a **Minimum** value of `200`, and enter a **Maximum** value of `800`.
7. Click the **Code** tab.
8. Add the following code at the bottom of the function in the **Get code** pane:

```
ie = obj.Interface;
propertyValue = ie.Height;
```

9. Add the following code at the bottom of the function in the **Set code** pane:

```
ie = get(obj, 'Interface');
set(ie, 'Height', propertyValue);
```

10. Save your driver.

Defining Functions

A common function for Internet Explorer is to download a Web page. In the following steps, you create a function called `goTo` that allows you to navigate the Web with the browser.

1. Select the `Functions` node in the driver editor tree.
2. In the **Add function** field, enter the text `goTo`, and click **Add**.
3. Expand the `Functions` node in the tree, and select the new node `goTo`.

   Writing functions for generic drivers in the MATLAB Instrument Driver Editor is a matter of writing straight code.

   Your `goTo` function requires only one input argument: the URL of the Web page to navigate to. You can call that argument `site`.
4. Change the first line of the MATLAB code pane to read

```
function goTo(obj, site)
```

   The variable `obj` is the device object using this driver. The value of `site` is a character vector passed into this function when you are using this driver. Your function then must pass the value of `site` on to the IE COM object. So your function must get a handle to the COM object, then call the IE COM method `Navigate2`, passing to it the value of `site`.
5. Add the following code at the bottom of the function in the MATLAB code pane:

```
ie = obj.Interface;
invoke(ie, 'Navigate2', site);
```

6. Save your driver, and close the MATLAB Instrument Driver Editor.

   Now that your generic driver is ready, you can use it with the **Test and Measurement Tool** or at the MATLAB command line.

Using Generic Driver with Test & Measurement Tool

In this section...

"Creating and Connecting the Device Object" on page 16-7
"Accessing Properties" on page 16-8
"Using Functions" on page 16-8

Creating and Connecting the Device Object

With the Test & Measurement Tool you can scan for your driver, create a device object that uses that driver, set and get properties of the object, and execute functions.

This example illustrates how to use the generic driver you created in "Writing a Generic Driver" on page 16-3.

1. If your driver is not in the *matlabroot*`\toolbox\instrument\instrument\drivers` directory, in the MATLAB Command Window, make sure that the directory containing your driver is on the MATLAB path.

   ```
   path
   ```

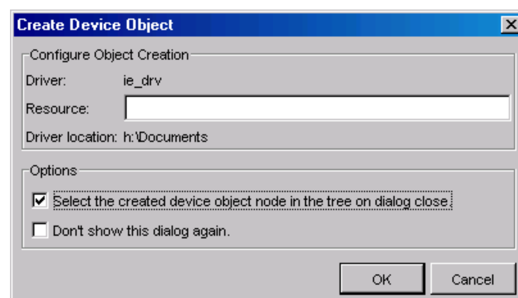
   If you do not see the directory in the path listing, and the driver is not in the *matlabroot*`\toolbox\instrument\instrument\drivers` directory, add the directory to the path with the command

   ```
   addpath directory
   ```

   where *directory* is the pathname to the directory containing your driver.
2. Open the Test & Measurement Tool.

   ```
   tmtool
   ```

3. In the Test & Measurement Tool tree, expand the `Instrument Drivers` node.
4. Select the `MATLAB Instrument Drivers` node.
5. Your driver might not be listed yet, so click **Scan** in the lower-right corner of the tool. If the tool found your driver, it is listed in the tree as `ie_drv.mdd`.
6. Select the `ie_drv.mdd` node in the tree.
7. Right-click the `ie_drv.mdd` node in the tree, and select **Create Device Object Using Driver**. The following dialog box appears.

8. Select the **Select the created device object in the tree on dialog close** check box. The device object in this example does not need a resource, so keep that field empty.

9 Click **OK**.

When the Test & Measurement Tool creates the device object, an entry for the object appears as a node in the tree. The `Browser-ie_drv` node should already be selected in the tree. This refers to the device object you just created.

10 Click **Connect** in the upper-right corner of the Test & Measurement Tool. This establishes a communication channel between the tool and the IE browser window, and an empty IE window appears on your screen. Remember that the **Create** code for your driver creates an object for the IE browser, and the **Connect** code and makes its window visible.

Accessing Properties

The driver you created allows you to specify where the browser window appears on your screen and how large it is.

1 Click the **Properties** tab, and then select `Top` in the **Device object properties** list.

The first value displayed for setting this property is `0.0`.

2 Click **Set**. The IE browser window shifts upward to the top edge of your screen.
3 With the mouse, grab the IE window, and drag it down some distance from the top of the screen.
4 Now return to the Test & Measurement Tool window, and click **Get** for the `Top` property. Notice in the **Response** pane how many pixels down you have moved the window.

Use your driver `Vsize` property to change the size of the browser window.

1 Select `Vsize` in the **Device object properties** list.
2 Enter a property value of `200`, and click **Set**. Notice the IE window size.
3 Enter a property value of `400` and click **Set**. Notice the IE window size.
4 Try resizing the IE browser window directly with the mouse. Then in the Test & Measurement Tool, click **Get** for the `Vsize` property. Notice the value returned to the **Response** pane.

Using Functions

Use the `goTo` function of your generic driver to control the Web page that the browser displays.

1 In the Test & Measurement Tool, click the **Functions** tab for your device object.
2 Select `goTo` in the list of **Device object functions**.
3 In the **Input argument(s)** field, enter `'www.mathworks.com'`. Be sure to include the single quotes.
4 Click **Execute**. Observe the IE browser and see that it displays the MathWorks Web site.
5 Experiment freely. When you are finished, right-click the `Browser-ie_drv node` in the tree and select **Delete Object**. Close the Test & Measurement Tool, and close the IE browser window you created in this example.

Using a Generic Driver at Command Line

In this section...

"Creating and Connecting the Device Object" on page 16-9
"Accessing Properties" on page 16-9
"Using Functions" on page 16-10

Creating and Connecting the Device Object

The Instrument Control Toolbox software provides MATLAB commands you can use in the Command Window or in files to create a device object that uses a driver, set and get properties of the object, and execute functions.

This example illustrates how to use the generic driver you created in "Writing a Generic Driver" on page 16-3.

1 If your driver is not in the `*matlabroot*\toolbox\instrument\instrument\drivers` directory, in the MATLAB Command Window, make sure that the directory containing your driver is on the MATLAB software path.

```
path
```

If you do not see the directory in the path listing, and the driver is not in the *matlabroot*`\toolbox\instrument\instrument\drivers` directory, add the directory to the path with the command

```
addpath directory
```

where *directory* is the pathname to the directory containing your driver.

2 Create a device object using your driver. For the driver used in this example, the `icdevice` function does not require an argument for a resource when using a generic driver. What the object connects to and how it makes that connection are defined in the **Create** code of your driver.

```
ie_obj = icdevice('ie_drv');
```

3 Connect the object.

```
connect(ie_obj);
```

When the device object is connected, an empty IE window appears on your screen. Now you can communicate directly with the IE browser from the MATLAB Command window.

Accessing Properties

The driver you created allows you to specify where the browser window appears on your screen and how large it is. You read and write the properties of your device object with the `get` and `set` functions, respectively.

1 View all of the properties of your device object.

```
get(ie_obj)
    ConfirmationFcn =
    DriverName = ie_drv.mdd
```

```
DriverType = MATLAB generic
InstrumentModel =
Interface = [1x1 COM.internetexplorer_application]
LogicalName =
Name = Browser-ie_drv
ObjectVisibility = on
RsrcName =
Status = open
Tag =
Timeout = 10
Type = Browser
UserData = []

BROWSER specific properties:
Top = 47
Vsize = 593
```

2 Most of the properties listed belong to all device objects. For this example, the properties of interest are those listed as `BROWSER specific properties`, that is, `Top` and `Vsize`.

The `Top` property defines the IE browser window position in pixels from the top of the screen. `Vsize` defines the vertical size of the window in pixels.

3 Shift the IE browser window to the top of the screen.

```
ie_obj.Top = 0;
```

4 With the mouse, grab and drag the IE browser window down away from the top of the screen.

5 Find the window's new position by examining the `Top` property.

```
ie_obj.Top
ans =
   120
```

Adjust the size of the window by setting the `Vsize` property.

```
ie_obj.Vsize = 200);
```

6 Make the window larger by increasing the property value.

```
ie_obj.Vsize = 600);
```

Using Functions

By using the `goTo` function of your generic driver, you can control the Web page displayed in the IE browser window.

1 View all of the functions (methods) of your device object.

```
methods(ie_obj)

Methods for class icdevice:

Contents    disp       icdevice      instrnotify methods   size
class       display    igetfield     instrument  ne        subsasgn
close       end        inspect       invoke      obj2mfile subsref
connect     eq         instrcallback isa         open      vertcat
ctranspose  fieldnames instrfind     isequal     openvar
delete      get        instrfindall  isetfield   propinfo
devicereset geterror   instrhelp     isvalid     selftest
disconnect  horzcat    instrhwinfo   length      set

Driver specific methods for class icdevice:
```

```
goTo
```

Most of the methods listed apply to all device objects. For this example, the method of interest is the one listed under `Driver specific methods`, that is, `goTo`.

2 Use the `goTo` function to specify the page for the IE browser to display.

```
invoke(ie_obj, 'goTo', 'www.mathworks.com');
```

If you have access to the Internet, the IE window should display the MathWorks Web site.

3 When you are finished with your example, clean up the MATLAB workspace by removing the object.

```
disconnect(ie_obj);
delete(ie_obj);
clear ie_obj;
```

4 Close the IE browser window you created in this example.

17

Saving and Loading the Session

This chapter describes how to save and load information associated with an instrument control session.

- "Saving and Loading Instrument Objects" on page 17-2
- "Debugging: Recording Information to Disk" on page 17-5

Saving and Loading Instrument Objects

In this section...
"Saving Instrument Objects to a File" on page 17-2
"Saving Objects to a MAT-File" on page 17-3

Saving Instrument Objects to a File

You can save an instrument object to a file using the `obj2mfile` function. `obj2mfile` provides you with these options:

- Save all property values or save only those property values that differ from their default values.

  Read-only property values are not saved. Therefore, read-only properties use their default values when you load the instrument object into the MATLAB workspace. To determine if a property is read-only, use the `propinfo` function or examine the property reference pages.
- Save property values using the `set` syntax or the dot notation.

If the `UserData` property is not empty, or if a callback property is set to a cell array of values or a function handle, then the data stored in these properties is written to a MAT-file when the instrument object is saved. The MAT-file has the same name as the file containing the instrument object code.

For example, suppose you create the GPIB object `g`, return instrument identification information to the variable `out`, and store `out` in the `UserData` property.

```
g = gpib('ni',0,1);
g.Tag = 'My GPIB object';
fopen(g)
cmd = '*IDN?';
fprintf(g,cmd)
out = fscanf(g);
g.UserData = out;
```

The following command saves `g` and the modified property values to the file `mygpib.m`. Because the `UserData` property is not empty, its value is automatically written to the MAT-file `mygpib.mat`.

```
obj2mfile(g,'mygpib.m');
```

Use the `type` command to display `mygpib.m` at the command line.

Loading the Instrument Object

To load an instrument object that was saved as a file into the MATLAB workspace, type the name of the file at the command line. For example, to load `g` from the file `mygpib.m`,

```
g = mygpib
```

The display summary for `g` is shown below. Note that the read-only properties such as `Status`, `BytesAvailable`, `ValuesReceived`, and `ValuesSent` are restored to their default values.

```
GPIB Object Using NI Adaptor : GPIB0-1

Communication Address
   BoardIndex:         0
```

```
   PrimaryAddress:     1
   SecondaryAddress:   0

Communication State
   Status:             closed
   RecordStatus:       off

Read/Write State
   TransferStatus:     idle
   BytesAvailable:     0
   ValuesReceived:     0
   ValuesSent:         0
```

When loading `g` into the workspace, the MAT-file `mygpib.mat` is automatically loaded and the `UserData` property value is restored.

```
g.UserData
ans =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

Saving Objects to a MAT-File

You can save an instrument object to a MAT-file just as you would any workspace variable — using the `save` command. For example, to save the GPIB object `g` and the variables `cmd` and `out`, defined in “Saving Instrument Objects to a File” on page 17-2, to the MAT-file `mygpib1.mat`,

```
save mygpib1 g cmd out
```

Read-only property values are not saved. Therefore, read-only properties use their default values when you load the instrument object into the MATLAB workspace. To determine if a property is read-only, use the `propinfo` function or examine the property reference pages.

Loading the Instrument Object

To load an instrument object that was saved to a MAT-file into the MATLAB workspace, use the `load` command. For example, to load `g`, `cmd`, and `out` from MAT-file `mygpib1.mat`,

```
load mygpib1
```

The display summary for `g` is shown below. Note that the read-only properties such as `Status`, `BytesAvailable`, `ValuesReceived`, and `ValuesSent` are restored to their default values.

```
GPIB Object Using NI Adaptor : GPIB0-1

Communication Address
   BoardIndex:         0
   PrimaryAddress:     1
   SecondaryAddress:   0

Communication State
   Status:             closed
   RecordStatus:       off

Read/Write State
   TransferStatus:     idle
   BytesAvailable:     0
```

```
ValuesReceived:     0
ValuesSent:         0
```

Debugging: Recording Information to Disk

In this section...

"Using the record Function" on page 17-5
"Introduction to Recording Information" on page 17-5
"Creating Multiple Record Files" on page 17-6
"Specifying a File Name" on page 17-6
"Record File Format" on page 17-6
"Recording Information to Disk" on page 17-7

Using the record Function

Recording information to disk provides a permanent record of your instrument control session, and is an easy way to debug your application. While the instrument object is connected to the instrument, you can record this information to a disk file:

- The number of values written to the instrument, the number of values read from the instrument, and the data type of the values
- Data written to the instrument, and data read from the instrument
- Event information

You record information to a disk file with the `record` function. The properties associated with recording information to disk are given below.

Recording Properties

Property Name	Description
RecordDetail	Specify the amount of information saved to a record file.
RecordMode	Specify whether data and event information are saved to one record file or to multiple record files.
RecordName	Specify the name of the record file.
RecordStatus	Indicate if data and event information are saved to a record file.

Introduction to Recording Information

This example creates the GPIB object `g`, records the number of values transferred between `g` and the instrument, and stores the information to the file text `myfile.txt`.

```
g = gpib('ni',0,1);
g.RecordName = 'myfile.txt';
fopen(g)
record(g)
fprintf(g,'*IDN?')
out = fscanf(g);
```

End the instrument control session.

```
fclose(g)
delete(g)
clear g
```

Use the `type` command to display `myfile.txt` at the command line.

Creating Multiple Record Files

When you initiate recording with the `record` function, the `RecordMode` property determines if a new record file is created or if new information is appended to an existing record file.

You can configure `RecordMode` to `overwrite`, `append`, or `index`. If `RecordMode` is `overwrite`, then the record file is overwritten each time recording is initiated. If `RecordMode` is `append`, then the new information is appended to the file specified by `RecordName`. If `RecordMode` is `index`, a different disk file is created each time recording is initiated. The rules for specifying a record file name are discussed in "Specifying a File Name" on page 17-6.

Specifying a File Name

You specify the name of the record file with the `RecordName` property. You can specify any value for `RecordName`, including a directory path, provided the file name is supported by your operating system. Additionally, if `RecordMode` is `index`, then the file name follows these rules:

- Indexed file names are identified by a number. This number precedes the file name extension and is increased by 1 for successive record files.
- If no number is specified as part of the initial file name, then the first record file does not have a number associated with it. For example, if `RecordName` is `myfile.txt`, then `myfile.txt` is the name of the first record file, `myfile01.txt` is the name of the second record file, and so on.
- `RecordName` is updated after the record file is closed.
- If the specified file name already exists, then the existing file is overwritten.

Record File Format

The record file is an ASCII file that contains a record of one or more instrument control sessions. You specify the amount of information saved to a record file with the `RecordDetail` property.

`RecordDetail` can be `compact` or `verbose`. A compact record file contains the number of values written to the instrument, the number of values read from the instrument, the data type of the values, and event information. A verbose record file contains the preceding information as well as the data transferred to and from the instrument.

Binary data with precision given by `uchar`, `schar`, (u)`int8`, (u)`int16`, or (u)`int32` is recorded as hexadecimal values. For example, if the integer value 255 is read from the instrument as a 16-bit integer, the hexadecimal value 00FF is saved in the record file. Single- and double-precision floating-point numbers are recorded as decimal values using the `%g` format, and as hexadecimal values using the format specified by the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.

The IEEE floating-point format includes three components — the sign bit, the exponent field, and the significant field. Single-precision floating-point values consist of 32 bits, and the value is given by

$$\text{value} = (-1)^{\text{sign}}(2^{\text{exp-127}})(1.\text{significand})$$

Double-precision floating-point values consist of 64 bits, and the value is given by

```
value = (-1)^sign(2^(exp-1023))(1.significand)
```

The floating-point format component and the associated single-precision and double-precision bits are given below.

Format Component	Single-Precision Bits	Double-Precision Bits
sign	1	1
exp	2-9	2-12
significand	10-32	13-64

For example, suppose you record the decimal value 4.25 using the single-precision format. The record file stores 4.25 as the hex value 40880000, which is calculated from the IEEE single-precision floating-point format. To reconstruct the original value, convert the hex value to a decimal value using `hex2dec`:

```
dval = hex2dec('40880000')
dval =
    1.082654720000000e+009
```

Convert the decimal value to a binary value using `dec2bin`:

```
bval = dec2bin(dval,32)
bval =
01000000100010000000000000000000
```

The interpretation of `bval` is given by the preceding table. The left most bit indicates the value is positive because $(-1)^0 = 1$. The next 8 bits correspond to the exponent, which is given by

```
exp = bval(2:9)
exp =
10000001
```

The decimal value of `exp` is $2^7+2^0 = 129$. The remaining bits correspond to the significant, which is given by

```
significand = bval(10:32)
significand =
00010000000000000000000
```

The decimal value of `significand` is $2^{-4} = 0.0625$. You reconstruct the original value by plugging the decimal values of `exp` and `significand` into the formula for IEEE singles:

```
value = (-1)^0(2^(129 - 127))(1.0625)
value = 4.25
```

Recording Information to Disk

This example extends “Writing and Reading Binary Data” on page 4-12 by recording the associated information to a record file. Additionally, the structure of the resulting record file is presented:

1 **Create an instrument object** — Create the GPIB object `g` associated with a National Instruments GPIB controller with board index 0, and an instrument with primary address 1.

```
g = gpib('ni',0,1);
```

2 **Configure properties** — Configure the input buffer to accept a reasonably large number of bytes, and configure the timeout value to two minutes to account for slow data transfer.

```
g.InputBufferSize = 50000;
g.Timeout = 120;
```

Configure `g` to execute the callback function `instrcallback` every time 5000 bytes are stored in the input buffer.

```
g.BytesAvailableFcnMode = 'byte';
g.BytesAvailableFcnCount = 5000;
g.BytesAvailableFcn = @instrcallback;
```

Configure `g` to record information to multiple disk files using the verbose format. The first disk file is defined as `WaveForm1.txt`.

```
g.RecordMode = 'index';
g.RecordDetail = 'verbose';
g.RecordName = 'WaveForm1.txt';
```

3 **Connect to the instrument** — Connect `g` to the oscilloscope.

```
fopen(g)
```

4 **Write and read data** — Initiate recording.

```
record(g)
```

Configure the scope to transfer the screen display as a bitmap.

```
fprintf(g,'HARDCOPY:PORT GPIB')
fprintf(g,'HARDCOPY:FORMAT BMP')
fprintf(g,'HARDCOPY START')
```

Initiate the asynchronous read operation, and begin generating events.

```
readasync(g)
```

`instrcallback` is called every time 5000 bytes are stored in the input buffer. The resulting displays are shown below.

```
BytesAvailable event occurred at 09:04:33 for the object: GPIB0-1.
BytesAvailable event occurred at 09:04:42 for the object: GPIB0-1.
BytesAvailable event occurred at 09:04:51 for the object: GPIB0-1.
BytesAvailable event occurred at 09:05:00 for the object: GPIB0-1.
BytesAvailable event occurred at 09:05:10 for the object: GPIB0-1.
BytesAvailable event occurred at 09:05:19 for the object: GPIB0-1.
BytesAvailable event occurred at 09:05:28 for the object: GPIB0-1.
```

Wait until all the data is stored in the input buffer, and then transfer the data to the MATLAB workspace as unsigned 8-bit integers.

```
out = fread(g,g.BytesAvailable,'uint8');
```

Toggle the recording state from `on` to `off`. Because the `RecordMode` value is `index`, the record file name is automatically updated.

```
record(g)
g.RecordStatus
```

```
ans =
off
g.RecordName
ans =
WaveForm2.txt
```

5 **Disconnect and clean up** — When you no longer need `g`, you should disconnect it from the instrument, and remove it from memory and from the MATLAB workspace.

```
fclose(g)
delete(g)
clear g
```

The Record File Contents

To display the contents of the WaveForm1.txt record file,

```
type WaveForm1.txt
```

The record file contents are shown below. Note that data returned by the `fread` function is in hex format (most of the bitmap data is not shown).

```
Legend:
  * - An event occurred.
  > - A write operation occurred.
  < - A read operation occurred.

1     Recording on 18-Jun-2000 at 09:03:53.529. Binary data in
      little endian format.
2   > 18 ascii values.
      HARDCOPY:PORT GPIB
3   > 19 ascii values.
      HARDCOPY:FORMAT BMP
4   > 14 ascii values.
      HARDCOPY START
5   * BytesAvailable event occurred at 18-Jun-2000 at 09:04:33.334
6   * BytesAvailable event occurred at 18-Jun-2000 at 09:04:41.775
7   * BytesAvailable event occurred at 18-Jun-2000 at 09:04:50.805
8   * BytesAvailable event occurred at 18-Jun-2000 at 09:04:00.266
9   * BytesAvailable event occurred at 18-Jun-2000 at 09:05:10.306
10  * BytesAvailable event occurred at 18-Jun-2000 at 09:05:18.777
11  * BytesAvailable event occurred at 18-Jun-2000 at 09:05:27.778
12  < 38462 uint8 values.
      42 4d cf 03 00 00 00 00 00 00 3e 00 00 00 28 00
      00 00 80 02 00 00 e0 01 00 00 01 00 01 00 00 00
      00 00 00 96 00 00 00 00 00 00 00 00 00 00 00 00
      .
      .
      .
      ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
      ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
      ff ff ff ff ff ff ff ff ff ff ff ff ff ff
13  Recording off.
```

Test & Measurement Tool

This chapter describes how to use the Test & Measurement Tool to access your hardware interfaces and instrument drivers.

- "Test & Measurement Tool Overview" on page 18-2
- "Using the Test & Measurement Tool" on page 18-4

Test & Measurement Tool Overview

In this section...

"Instrument Control Toolbox Software Support" on page 18-2
"Navigating the Tree" on page 18-2

Instrument Control Toolbox Software Support

The Test & Measurement Tool (tmtool) enables you to configure and control resources (instruments, serial devices, drivers, interfaces, etc.) accessible through the toolbox without having to write the MATLAB script.

You can use the Test & Measurement Tool to manage your session with the toolbox. This tool enables you to do the following:

- Detect available hardware and drivers.
- Connect to an instrument or device.
- Configure instrument or device settings.
- Read and write data.
- Automatically generate the MATLAB script.
- Visualize acquired data.
- Export acquired data to the MATLAB workspace.

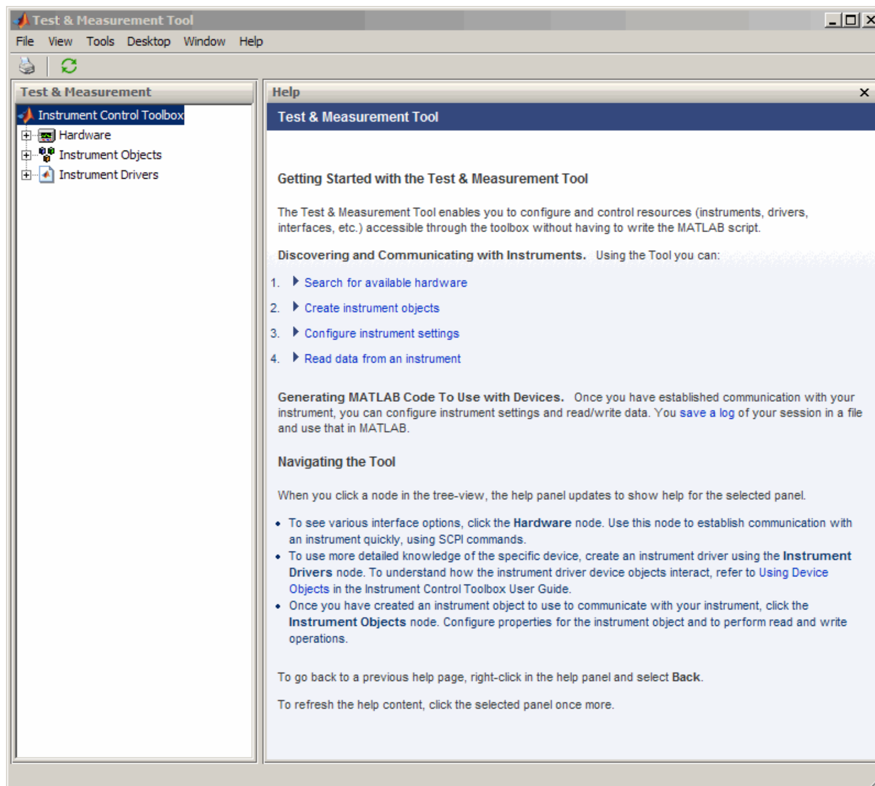
Navigating the Tree

You start the Test & Measurement Tool by typing

```
tmtool
```

You navigate to the various hardware control panes using the tool's tree. Start by selecting the toolbox you want to work with, which displays a set of instructions in the right pane. These instructions explain the basic steps to establishing communication with an instrument.

For example, the following figure shows the pane displayed when you select Instrument Control Toolbox.


Test & Measurement Tool
File View Tools Desktop Window Help
Test & Measurement
Instrument Control Toolbox
Hardware
Instrument Objects
Instrument Drivers
Help
Test & Measurement Tool
Getting Started with the Test & Measurement Tool
The Test & Measurement Tool enables you to configure and control resources (instruments, drivers, interfaces, etc.) accessible through the toolbox without having to write the MATLAB script.
Discovering and Communicating with Instruments. Using the Tool you can:
1. Search for available hardware
2. Create instrument objects
3. Configure instrument settings
4. Read data from an instrument
Generating MATLAB Code To Use with Devices. Once you have established communication with your instrument, you can configure instrument settings and read/write data. You save a log of your session in a file and use that in MATLAB.
Navigating the Tool
When you click a node in the tree-view, the help panel updates to show help for the selected panel.
To see various interface options, click the Hardware node. Use this node to establish communication with an instrument quickly, using SCPI commands.
To use more detailed knowledge of the specific device, create an instrument driver using the Instrument Drivers node. To understand how the instrument driver device objects interact, refer to Using Device Objects in the Instrument Control Toolbox User Guide.
Once you have created an instrument object to use to communicate with your instrument, click the Instrument Objects node. Configure properties for the instrument object and to perform read and write operations.
To go back to a previous help page, right-click in the help panel and select Back.
To refresh the help content, click the selected panel once more.

Using the Test & Measurement Tool

In this section...

"Overview of the Examples" on page 18-4
"Hardware" on page 18-4
"Instrument Objects" on page 18-9
"Instrument Drivers" on page 18-13

Overview of the Examples

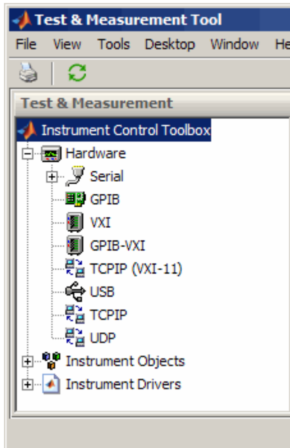
These examples illustrates a typical session using the Test & Measurement Tool for instrument control. The session entails communicating with a Tektronix TDS 210 oscilloscope via a GPIB interface.

To start the tool, on the MATLAB Command window, type:

```
tmtool
```

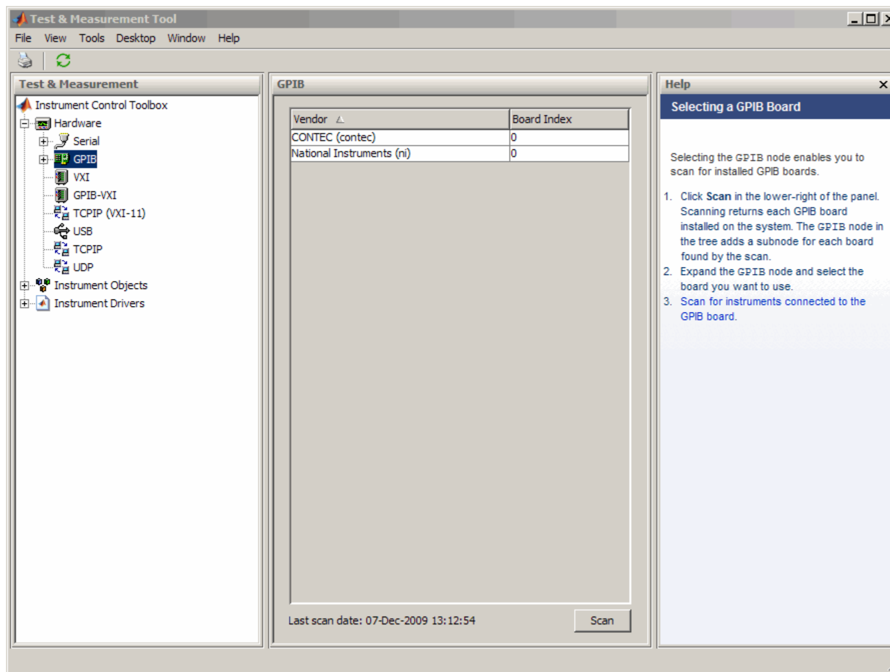
Hardware

When the tool displays, expand (click the +) the `Instrument Control Toolbox` node in the tree. Next, expand the `Hardware` node. The tree now looks like this.



Selecting the Interface and Scanning for GPIB Boards

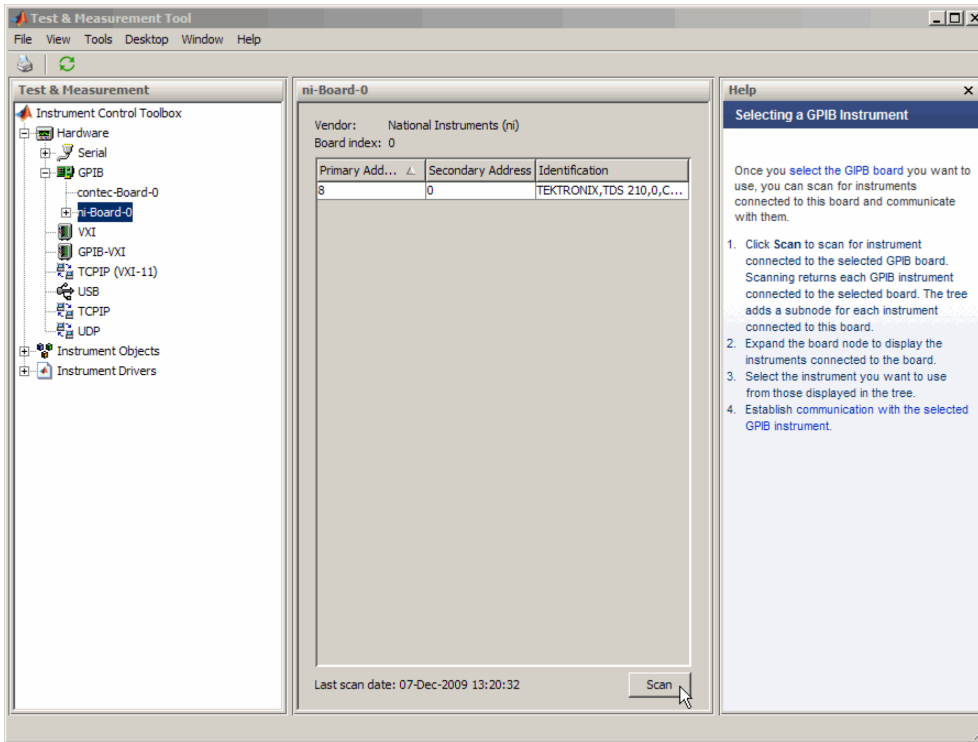
Next, scan for installed GPIB boards by selecting the `GPIB` node. The right pane changes to the **Installed GPIB Board** list. Click **Scan** to see what boards are installed. The following figure shows the scan result from a system with one Capital Equipment Corp and one Keithley® GPIB board.



Scanning for Instruments Connected to GPIB Boards

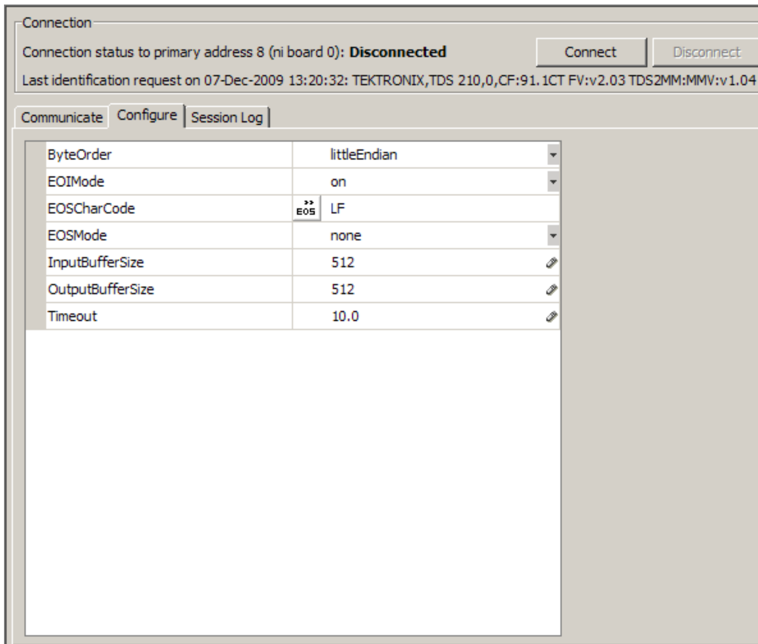
After determining what GPIB boards are installed, you must determine what instruments are connected to those boards. Expand the `GPIB` node and select a board.

The right pane changes to the **GPIB Instruments** list. Click **Scan** to see what instruments are connected to this board. The following figure shows the scan result from a system with a Tektronix TDS 210 connected at primary address 8`.



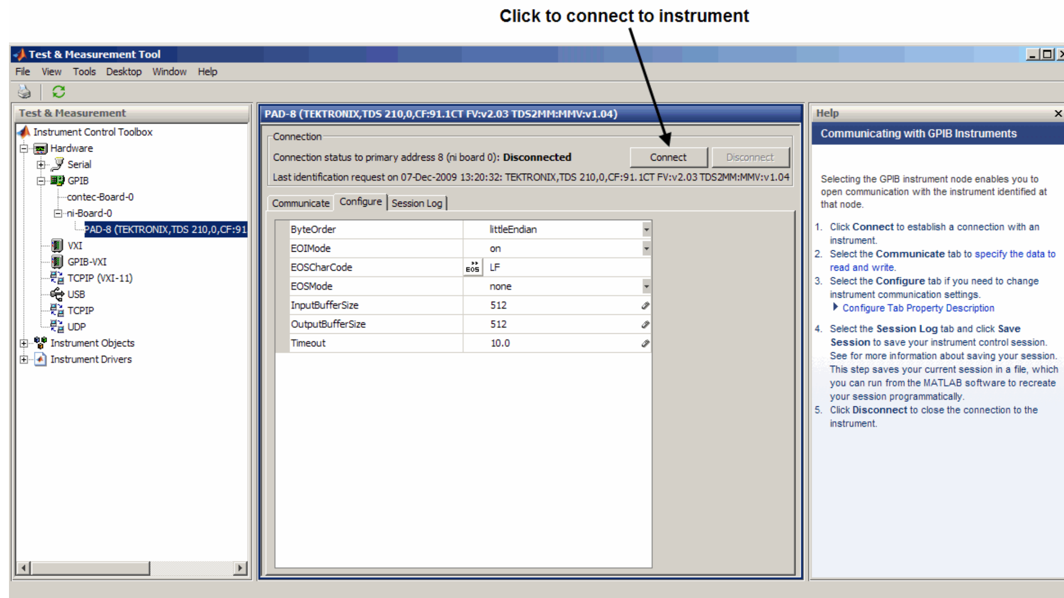
Configuring the Interface

You can change the configuration of the interface by clicking the **Configure** tab. This pane displays properties you can set to configure the instrument communication settings. In the following view of the **Configure** pane, the `Timeout` property value has been set to `10` seconds.



Establishing the Connection

Expand the `ni-Board-0` node and select the instrument at primary address 4: `PAD-8 (TEKTRONIX,TDS 210...)`. The right pane changes to the control pane you use for writing and reading data to and from that instrument.



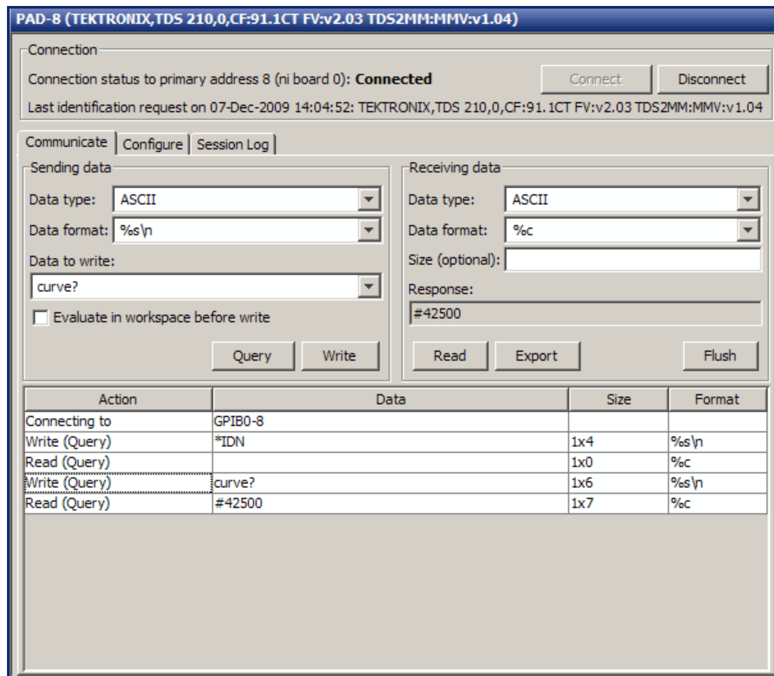
Click **Connect** to establish communication with the instrument. The tool creates an interface object representing the communication channel to the instrument.

Writing and Reading Data

Selecting the **Communicate** tab displays the pane you use to write and read data. You can write and read data separately using the **Write** and **Read** buttons, or you can use the **Query** button to write and read in a single operation.

The following figure shows the pane after a brief session involving the following steps:

1. Open communication with the instrument.
2. Enter `*IDN?` as **Data to Write**, and click **Query** (write/read). This executes the identify command.
3. Enter `CURVE?` as **Data to Write**, and click **Query**. This retrieves the waveform data from the scope.

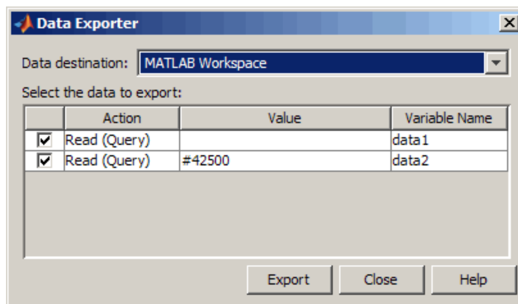


Exporting Instrument Data

You can export the data acquired from instruments to any of the following:

- MATLAB workspace as a variable
- Figure window as a plot
- MAT-file for storage in a file
- The MATLAB Variables editor for modification

To export data, select **File > Export > Instrument Response(s)** from the menu bar. When the Data Exporter dialog box opens, choose the variables to export. The following figure shows the Data Exporter set to export the curve data to the MATLAB workspace as the variable `data2`.



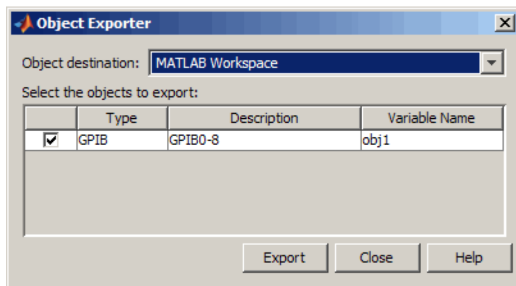
Note If you repeatedly generate a large amount of data in the Test and Measurement tool, you must delete the data object after you export it to MATLAB. This will allow the tool to return resources to MATLAB correctly and will prevent MATLAB from failing to respond the next time you acquire data.

Exporting the GPIB Object

When you open a connection to an instrument, the Test & Measurement Tool creates an instrument object automatically. You can export the GPIB instrument object created in this example as any of the following:

- MATLAB workspace object that you can use as an argument in instrument control commands
- File containing the call to the GPIB constructor and the commands to set object properties
- MAT-file for storage in a file

To export the object, select **File** > **Export** > **Instrument Object** from the menu bar. When the Object Exporter dialog box opens, choose the object to export. The following figure shows the Object Exporter set to export the object to a file. (When you run that file, it creates a new object with the equivalent settings.)



Saving Your Instrument Control Session

The **Session Log** tab displays the code equivalent of your instrument control session. You can save this code to a file so that you can execute the same commands programmatically.

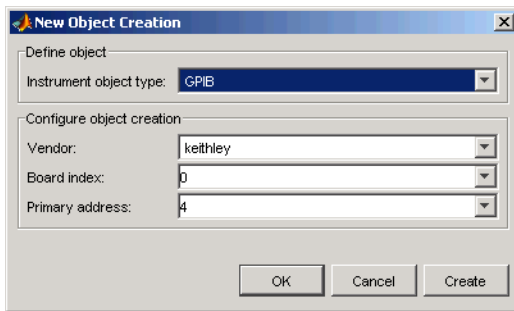
Select **File** > **Save Session Log** from the menu bar or click **Save Session**. From this dialog box you can specify a file name and directory location for the file.

Instrument Objects

Interface Objects

The Test & Measurement Tool creates interface objects automatically when you open a communication channel to an instrument by clicking the **Communication Status** button. To explicitly create and configure an interface object:

1. Expand the `Instrument Objects` node in the tree, and select `Interface Objects`. The **Interface Objects** pane appears on the right.
2. Click **New Object** to open the New Object Creation dialog box.

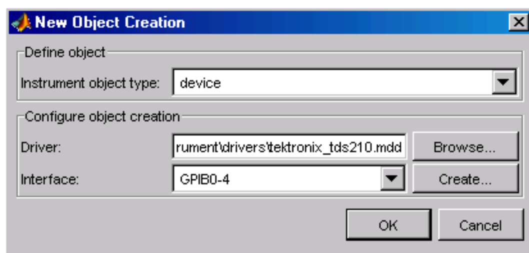


3 Specify the object parameters and click **OK** to create the new object.

Device Objects

To create and configure a device object:

1 Expand the `Instrument Objects` node in the tree, and select `Device Objects`. The **Device Objects** pane appears on the right.
2 Click **New Object** to open the New Object Creation dialog box. In this case, the **Instrument object type** is already set for `device`.

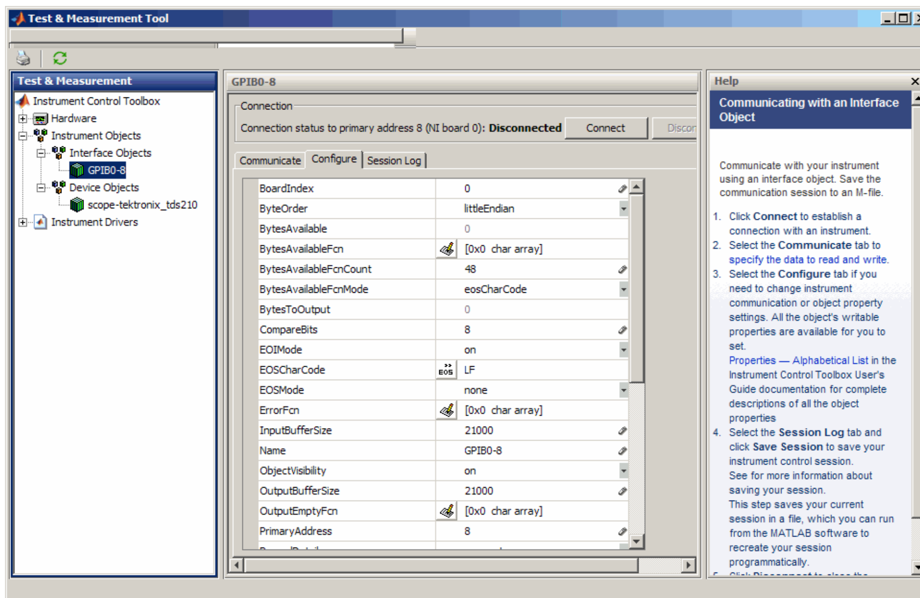


3 Specify or browse for the instrument driver you want to use; then choose from among the available interface objects, or create one if necessary.
4 Click **OK** to create the new device object.

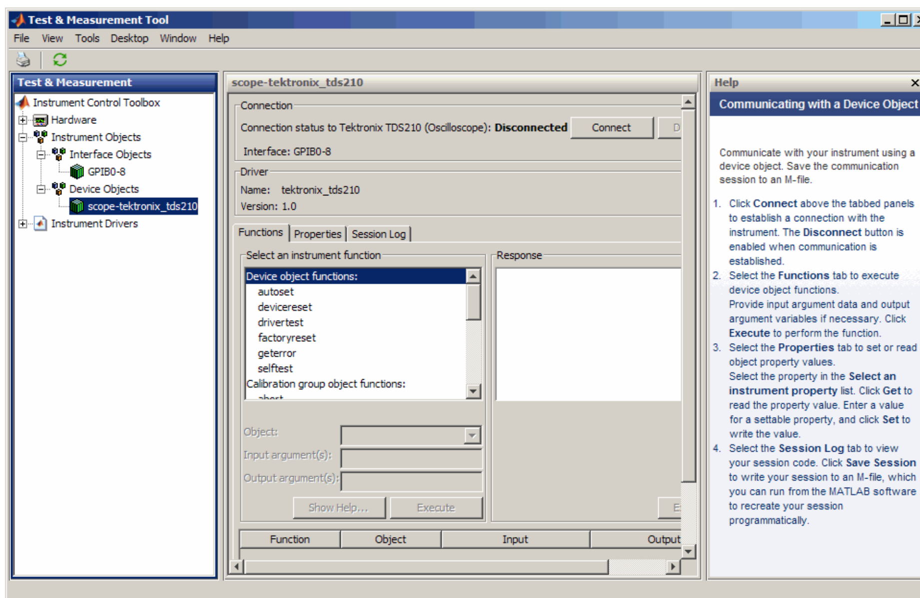
Setting Instrument Object Properties

Whether the instrument objects are created automatically, created through the New Object Creation dialog box, or created on the MATLAB Command window, the Test & Measurement Tool enables you to set the properties of these objects. To change object properties in the Test & Measurement Tool:

1 Expand the `Instrument Objects` node in the tree, then either `Interface Objects` or `Device Objects`, and select the object whose properties you want to set.
2 Click the **Configure** tab in the right pane.
3 Set properties displayed in this pane, as shown in the following figures.



Configuring Interface Object Properties

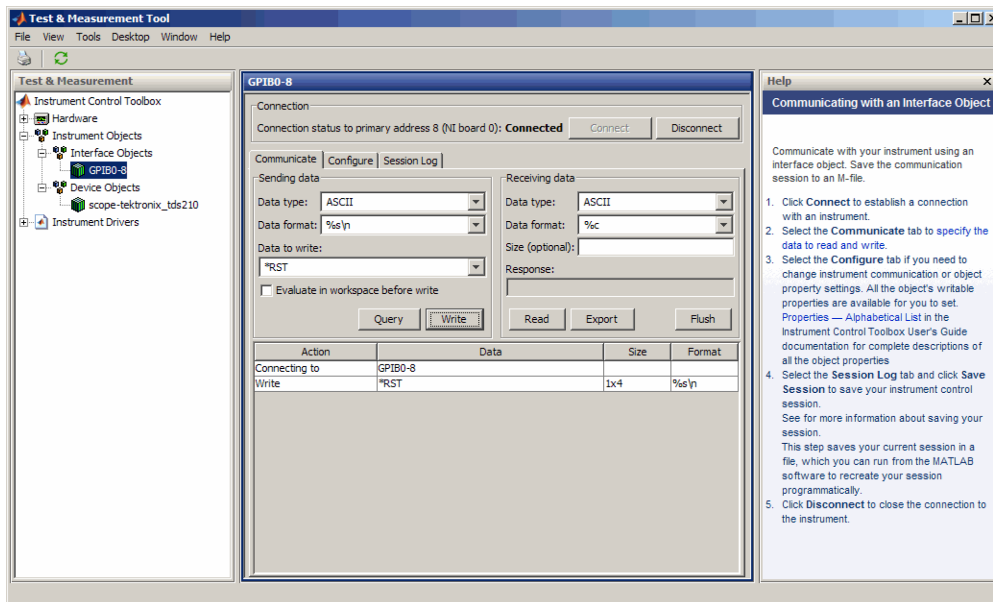


Configuring Device Object Properties

Communicating with Your Instrument

Using an Interface Object

When communicating with your instrument using an interface object, you send data to instrument in the form of raw instrument commands. In the following figure, the Test & Measurement Tool sends the *RST string to the TDS 210 oscilloscope via an interface object. *RST is the oscilloscope's reset command.

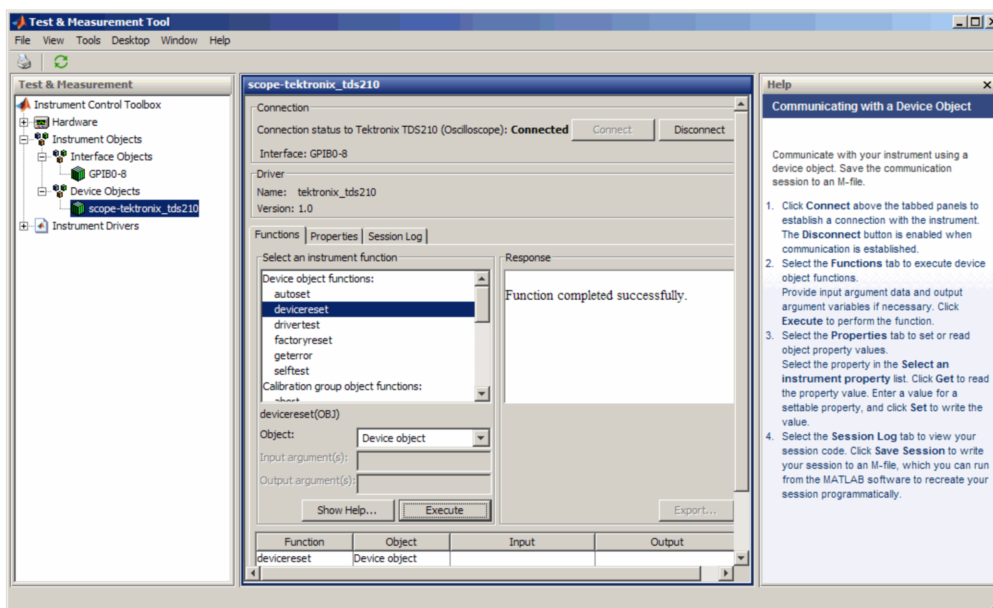


Communicating via an Interface Object

Using a Device Object

When communicating with your instrument using a device object, instead of employing instrument commands, you invoke device object methods (functions) or you set device object properties as provided by the MATLAB instrument driver for that instrument.

In the following figure, the Test & Measurement Tool resets a TDS 210 oscilloscope by issuing a call to the `devicereset` function of the instrument driver. Communicating this way, you don't need to know what the actual oscilloscope reset command is.



Communicating via a Device Object

Instrument Drivers

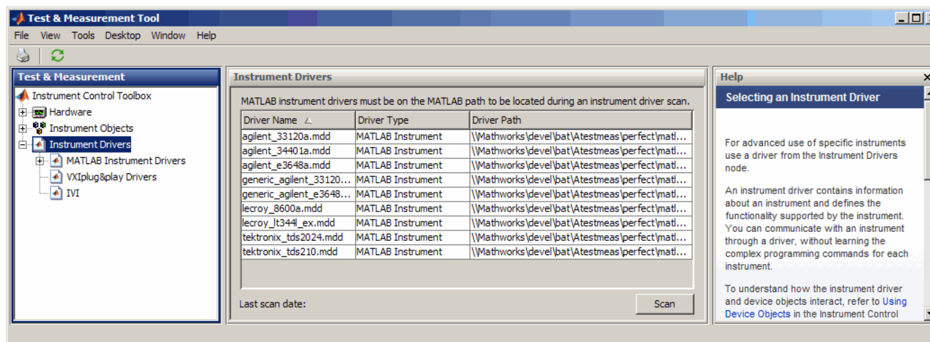
The Test & Measurement Tool enables you to scan for installed drivers, and to use those drivers when creating device objects.

MATLAB Instrument Drivers

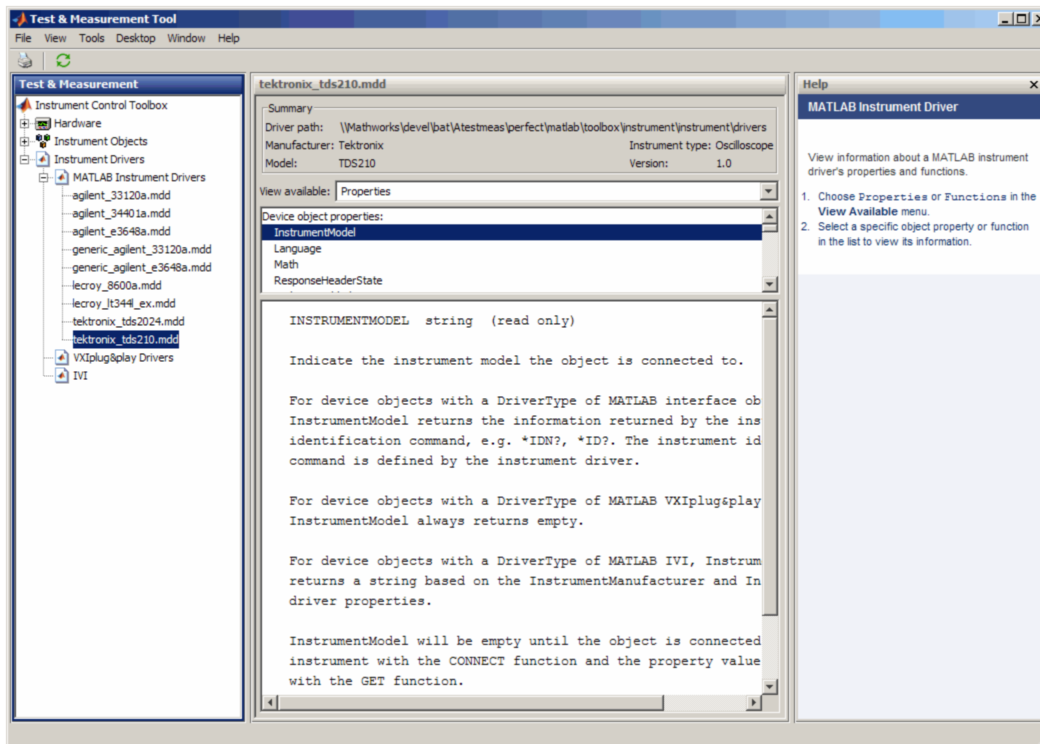
MATLAB instrument drivers include

- MATLAB interface drivers
- MATLAB VXI*plug&play* drivers
- MATLAB IVI drivers

Select the `MATLAB Instrument Drivers` node in the tree. Then click **Scan** to get an updated display of all the installed MATLAB instrument drivers found on the MATLAB software path.



When the Test & Measurement Tool scans for drivers, it makes them available as nodes under the driver type node. Expand the `MATLAB software Instrument Drivers` node to reveal the installed drivers. Select one of them to see the driver's details.



You can choose to see the driver's properties or functions. When you select the particular property or function, the tool displays that item's description.

VXIplug&play Drivers

For an example of scanning for installed VXI*plug&play* drivers with the Test & Measurement Tool, see “VXIplug&play Drivers” on page 13-3.

IVI Drivers

For an example of scanning for installed IVI-C drivers with the Test & Measurement Tool, see “Getting Started with IVI Drivers” on page 14-5. For using the Test & Measurement Tool to examine or configure an IVI configuration store, see “Configuring an IVI Configuration Store” on page 14-13.

Using the Instrument Driver Editor

This chapter describes how to use the Instrument Driver Editor to create, import, or modify instrument drivers.

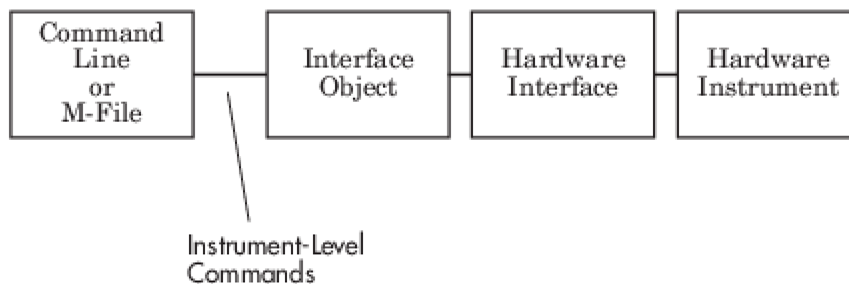
- “MATLAB Instrument Driver Editor Overview” on page 19-2
- “Creating MATLAB Instrument Drivers” on page 19-4
- “Properties” on page 19-13
- “Functions” on page 19-25
- “Groups” on page 19-33
- “Using Existing Drivers” on page 19-47

MATLAB Instrument Driver Editor Overview

In this section...
“What Is a MATLAB Instrument Driver?” on page 19-2
“How Does a MATLAB Instrument Driver Work?” on page 19-3
“Why Use a MATLAB Instrument Driver?” on page 19-3

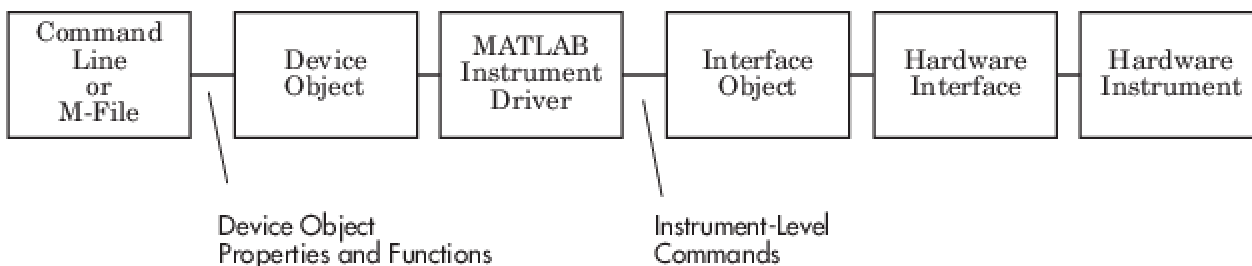
What Is a MATLAB Instrument Driver?

The Instrument Control Toolbox software provides the means of communicating directly with a hardware instrument through an interface object. If you are programming directly through an interface object, you need to program with the command language of the instrument itself. Any substitution of instrument, such as make or model, may require a change to the appropriate the MATLAB code.



A MATLAB instrument driver offers a layer of interpretation between you and the instrument. The instrument driver contains all the necessary commands for programming the instrument, so that you do not need to be aware of the specific instrument commands. Instead, you can program the instrument with familiar or consistent device object properties and functions.

The following figure shows how a device object and instrument driver offer a layer between the command line and the interface object. The instrument driver handles the instrument-level commands, so that as you program from the command line, you need only manipulate device object properties and functions, rather than instrument commands.



In addition to containing instrument commands, the instrument driver can also contain the MATLAB code to provide analysis based upon instrument setup or data.

Note For many instruments, a MATLAB instrument driver already exists and you will not need to create a MATLAB instrument driver for your instrument. For other instruments, there may be a

similar MATLAB instrument driver and you will need to edit it. If you would like more information on how to edit a MATLAB instrument driver, you may want to begin with "Modifying MATLAB Instrument Drivers" on page 19-47.

Note The Instrument Driver Editor is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

How Does a MATLAB Instrument Driver Work?

A MATLAB instrument driver contains information on the functionality supported by an instrument. You access this functionality through a device object's properties and functions.

When you query or configure a property of the device object using the `get` or `set` function, or when you call (`invoke`) a function on the device object, the MATLAB instrument driver provides a translation to determine what instrument commands are written to the instrument or what the MATLAB code is executed.

Why Use a MATLAB Instrument Driver?

Using a MATLAB instrument driver isolates you from the instrument commands. Therefore, you do not need to be aware of the instrument syntax, but can use the same code for a variety of related instruments, ignoring the differences in syntax from one instrument to the next.

For example, suppose you have two different oscilloscopes in your shop, each with its own set of commands. If you want to perform the same tasks with the two different instruments, you can create an instrument driver for each scope so that you can control each with the same code. Then substitution of one instrument for another does not require a change in the MATLAB code being used to control it, but only a substitution of the instrument driver.

Creating MATLAB Instrument Drivers

In this section...

"Driver Components" on page 19-4
"MATLAB Instrument Driver Editor Features" on page 19-4
"Saving MATLAB Instrument Drivers" on page 19-5
"Driver Summary and Common Commands" on page 19-5
"Initialization and Cleanup" on page 19-8

Driver Components

A MATLAB instrument driver contains information about an instrument and defines the functionality supported by the instrument.

Driver Component	Description
Driver Summary and Common Commands	Basic information about the instrument, e.g., manufacturer or model number.
Initialization and Cleanup	Code that is executed at various stages in the instrument control session, e.g., code that is executed upon connecting to the instrument.
Properties	A property is generally used to configure or query an instrument's state information.
Functions	A function is generally used to control or configure an instrument.
Groups	A group combines common functionality of the instrument into one component.

Depending on the instrument and the application for which the driver is being used, all components of the driver may not be defined. You can define the necessary driver components needed for your application with the MATLAB Instrument Driver Editor.

Note The Instrument Driver Editor is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

MATLAB Instrument Driver Editor Features

The MATLAB Instrument Driver Editor is a tool that creates or edits a MATLAB instrument driver. Specifically, it allows you to do the following:

- Add/remove/modify properties.
- Add/remove/modify functions.
- Define the MATLAB code to wrap around commands sent to instrument.

Open the MATLAB Instrument Driver Editor with the following command.

```
midedit
```

In the rest of this section, each driver component will be described and examples will be shown on how to add the driver component information to a new MATLAB instrument driver called `tektronix_tds210_ex.mdd`. The `tektronix_tds210_ex.mdd` driver will define basic information and instrument functionality for a Tektronix TDS 210 oscilloscope.

Saving MATLAB Instrument Drivers

You can save an instrument driver to any directory with any name. It is recommended that the instrument driver be saved to a directory in the MATLAB path and that the name follows the format `manufacturer_model.mdd`. For example, an instrument that is used with a Tektronix TDS 210 oscilloscope should be saved with the name `tektronix_tds210.mdd`.

Driver Summary and Common Commands

You can assign basic information about the instrument to the MATLAB instrument driver. Summary information can be used to identify the MATLAB instrument driver and the instrument that it represents. Common commands can be used to reset, test, and read error messages from the instrument. Together, this information can be used to initialize and verify the instrument.

Topics in this section include

- “Driver Summary” on page 19-5
- “Common Commands” on page 19-5
- “Defining Driver Summary and Common Commands” on page 19-6
- “Verifying Driver Summary and Common Commands” on page 19-6

Driver Summary

You can assign basic information that describes your instrument in the instrument driver. This information includes the manufacturer of the instrument, the model number of the instrument and the type of the instrument. A version can also be assigned to the driver to assist in revision control.

Common Commands

You can define basic common commands supported by the instrument. The common commands can be accessed through device object properties and functions.

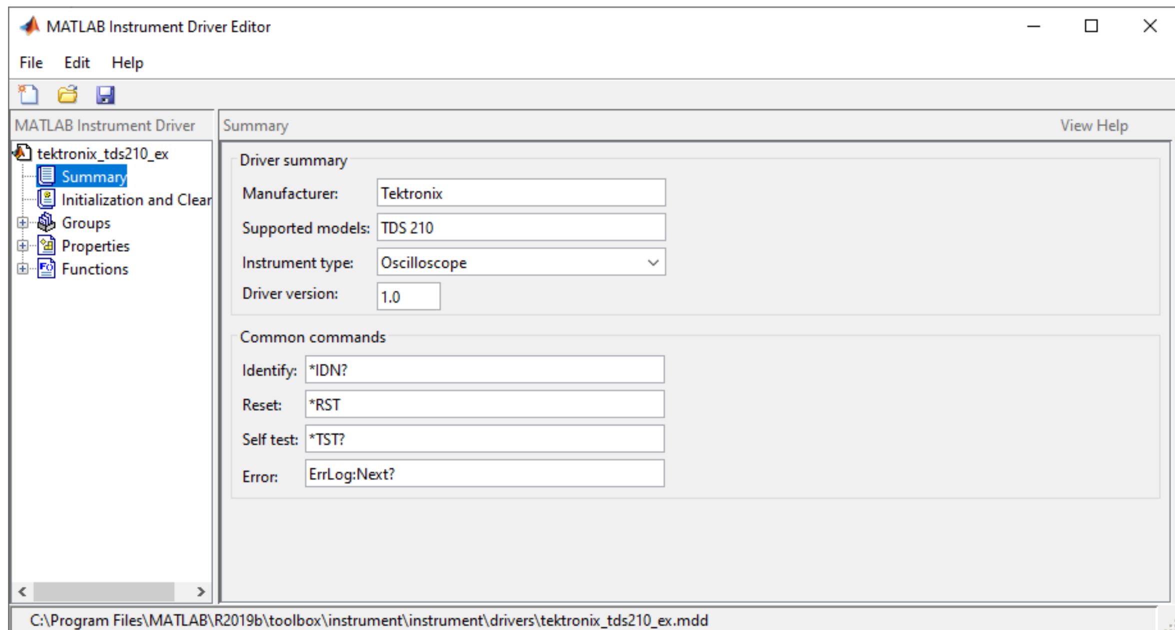
Common Commands	Accessed with Device Object's	Example Instrument Command	Description
Identify	`InstrumentModel` property	`*IDN?`	Returns the identification string of the instrument
Reset	`devicereset` function	`*RST`	Returns the instrument to a known state
Self test	`selftest` function	`*TST?`	Tests the instrument's interface
Error	`geterror` function	`ErrLog:Next?`	Retrieves the next instrument error message

The MATLAB Instrument Driver Editor assigns default values for the Common commands. The common commands should be modified appropriately to match the instrument's command set.

Defining Driver Summary and Common Commands

This example defines the basic driver information and Common commands for a Tektronix TDS 210 oscilloscope using the MATLAB Instrument Driver Editor:

1. Select the `Summary` node in the tree.
2. In the **Driver summary** pane:
   - a. Enter `Tektronix` in the **Manufacturer** field.
   - b. Enter `TDS 210` in the **Model** field.
   - c. Select `Oscilloscope` in the **Instrument type** field.
   - d. Enter `1.0` in the **Driver version** field.
3. In the **Common commands** pane:
   - a. Leave the **Identify** field with `*IDN?`.
   - b. Leave the **Reset** field with `*RST`.
   - c. Leave the **Self test** field with `*TST?`
   - d. Update the **Error** field with `ErrLog:Next?`
4. Click the **Save** button. Specify the name of the instrument driver as `tektronix_tds210_ex.mdd`.



Note For additional information on instrument driver nomenclature, refer to “Saving MATLAB Instrument Drivers” on page 19-5.

Verifying Driver Summary and Common Commands

This procedure verifies the summary information defined in the Driver Summary and Common commands panes. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB Command Window,

1 Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2 View the defined driver information.

```
obj
```

```
   Instrument Device Object Using Driver : tektronix_tds210_ex.mdd

   Instrument Information
      Type:               Oscilloscope
      Manufacturer:       Tektronix
      Model:              TDS210

   Driver Information
      DriverType:         MATLAB interface object
      DriverName:         tektronix_tds210_ex.mdd
      DriverVersion:      1.0

   Communication State
      Status:             closed
```

```
instrhwinfo(obj)
```

```
ans =

  struct with fields:

    Manufacturer: 'Tektronix'
           Model: 'TDS210'
            Type: 'Oscilloscope'
      DriverName: 'C:\Program Files\MATLAB\R2019b\toolbox\instrument\instrument\drivers\tektr
```

3 Connect to the instrument.

```
connect(obj)
```

4 Verify the Common commands.

```
obj.InstrumentModel
```

```
ans =

    TEKTRONIX,TDS 210,0,CF:91.1CT FV:v2.03 TDS2MM:MMV:v1.04
```

```
devicereset(obj)
```

```
selftest(obj)
```

```
ans =

    0
```

```
geterror(obj)
```

```
ans =
```

5 Disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

Initialization and Cleanup

This section describes how to define code that is executed at different stages in the instrument control session, so that the instrument can be set to a desired state at particular times. Specifically, you can define code that is executed after the device object is created, after the device object is connected to the instrument, or before the device object is disconnected from the instrument. Depending on the stage, the code can be defined as a list of instrument commands that will be written to the instrument or as MATLAB code.

Topics in this section include

- Definitions of the types of code that can be defined
- Examples of code for each supported stage
- Steps used to verify the code

Create Code

You define create code to ensure that the device object is configured to support the necessary properties and functions:

- Create code is evaluated immediately after the device object is created.
- Create code can only be defined as a MATLAB software code.

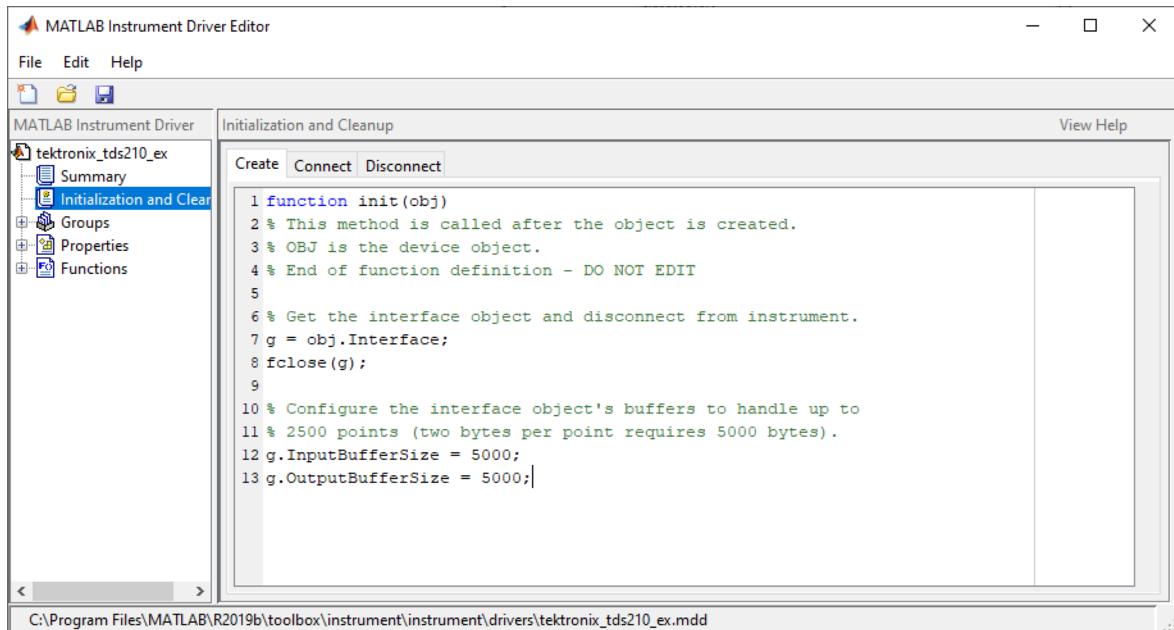
Defining Create Code

This example defines the create code that ensures that the device object can transfer the maximum waveform size, 2500 data points, supported by the Tektronix TDS 210 oscilloscope. In the MATLAB instrument driver editor,

1 Select the `Initialization and Cleanup` node in the tree.
2 Click the **Create** tab and enter the MATLAB software code to execute on device object creation.

```
% Get the interface object and disconnect from instrument.
g = obj.Interface;
fclose(g);

% Configure the interface object's buffers to handle up to
% 2500 points (two bytes per point requires 5000 bytes).
g.InputBufferSize = 5000;
g.OutputBufferSize = 5000;
```

3 Click the **Save** button.



Verifying Create Code

This procedure verifies the MATLAB software create code defined. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0.

1 From the MATLAB command line, create the interface object, `g`; and verify the default input and output buffer size values.

```
g = gpib('mcc',0,2);
g.InputBufferSize
```

```
ans =

   512
```

```
g.OutputBufferSize
```

```
ans =

   512
```

2 Create the device object, `obj`, using the `icdevice` function.

```
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

3 Verify the create code by querying the interface object's buffer sizes.

```
g.InputBufferSize
```

```
ans =

        5000
```

```
g.OutputBufferSize
```

```
ans =

        5000
```

4 Delete the objects.

```
delete([obj g])
```

Connect Code

In most cases you need to know the state or configuration of the instrument when you connect the device object to it. You can define connect code to ensure that the instrument is properly configured to support the device object's properties and functions.

Connect code is evaluated immediately after the device object is connected to the instrument with the `connect` function. The connect code can be defined as a series of instrument commands that will be written to the instrument or as MATLAB software code.

Defining Connect Code

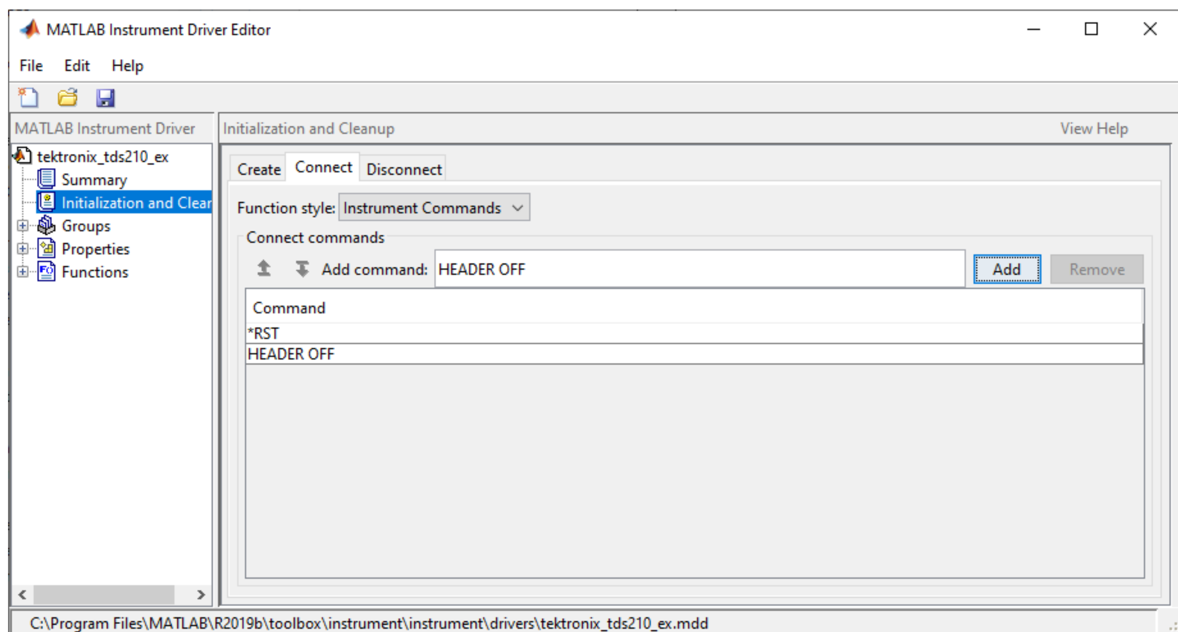
This example defines the connect code that ensures the Tektronix TDS 210 oscilloscope is configured to support the device object properties and functions. Specifically, the instrument will be returned to a known set of instrument settings (instrument reset) and the instrument will be configured to omit headers on query responses.

1 From the MATLAB instrument driver editor, select the `Initialization and Cleanup` node in the tree.

2 Click the **Connect** tab and enter the instrument commands to execute when the device object is connected to the instrument.

- Select `Instrument Commands` from the **Function style** menu.
- Enter the `*RST` command in the **Command** text field, and then click **Add**.
- Enter the `HEADER OFF` command in the **Command** text field, and then click **Add**.

3 Click the **Save** button.



Verifying Connect Code

This procedure verifies the instrument commands defined in the connect code. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0.

1. From the MATLAB command line, create the device object, `obj`, using the `icdevice` function.

   ```
   g = gpib('mcc',0,2);
   obj = icdevice('tektronix_tds210_ex.mdd',g);
   ```
2. Connect to the instrument.

   ```
   connect(obj)
   ```
3. Verify the connect code by querying the Header state of the instrument.

   ```
   query(g,'Header?')


   ans =

           0
   ```
4. Disconnect from the instrument and delete the objects.

   ```
   disconnect(obj)
   delete([obj g])
   ```

Disconnect Code

By defining disconnect code, you can ensure that the instrument and the device object are returned to a known state after communication with the instrument is completed.

Disconnect code is evaluated before the device object's being disconnected from the instrument with the `disconnect` function. This allows the disconnect code to communicate with the instrument. Disconnect code can be defined as a series of instrument commands that will be written to the instrument or it can be defined as MATLAB software code.

Defining Disconnect Code

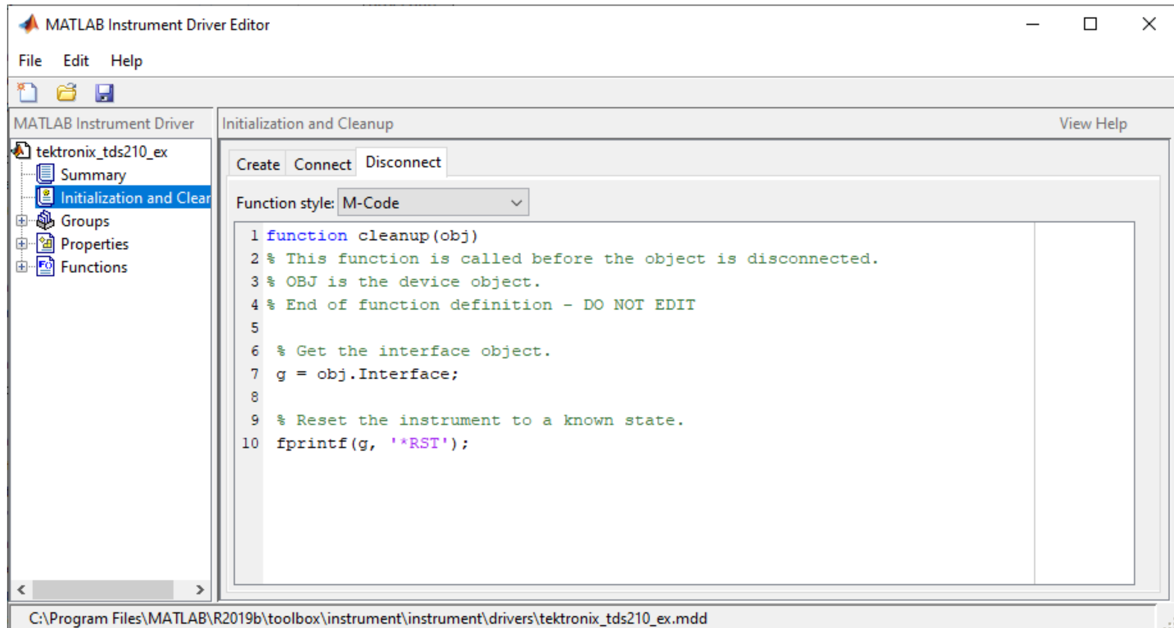
This example defines the disconnect code that ensures that the Tektronix TDS 210 oscilloscope is returned to a known state after communicating with the instrument using the device object.

1. From the MATLAB instrument driver editor, select the `Initialization and Cleanup` node in the tree.
2. Click the **Disconnect** tab and enter the MATLAB software code to execute when the device object is disconnected from the instrument.

   - Select `M-Code` from the **Function style** menu.
   - Define the MATLAB software code that will reset the instrument and configure the interface object's buffers to their default values.

     ```
     % Get the interface object.
     g = obj.Interface;

     % Reset the instrument to a known state.
     fprintf(g, '*RST');
     ```

3 Click the **Save** button.



Verifying Disconnect Code

This procedure verifies the MATLAB software code defined in the disconnect code. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2 Connect to the instrument.

```
connect(obj)
```

3 Alter some setting on the instrument so that a change can be observed when you disconnect. For example, the oscilloscope's contrast can be changed by pressing its front pane **Display** button, and then the **Contrast Decrease** button.

4 Disconnect from the instrument and observe that its display resets.

```
disconnect(obj)
```

5 Delete the objects.

```
delete([obj g])
```

Properties

In this section...
“Properties: Overview” on page 19-13
“Property Components” on page 19-13
“Examples of Properties” on page 19-15

Properties: Overview

You can make the programming of instruments through device objects easier and more consistent by using properties. A property can be used to query or set an instrument setting or attribute. For example, an oscilloscope's trigger level may be controlled with a property called `TriggerLevel`, which you can read or control with the `get` or `set` function. Even if two different scopes have different trigger syntax, you can use the same property name, `TriggerLevel`, to control them, because each scope will have its own instrument driver.

Another advantage of properties is that you can define them with certain acceptable values (enumerated) or limits (bounded) that can be checked before the associated commands are sent to the instrument.

Property Components

The behavior of the property is defined by the following components.

Set Code

The `set` code defines the code that is executed when the property is configured with the `set` function. The `set` code can be defined as an instrument command that will be written to the instrument or it can be defined as MATLAB software code.

If the `set` code is MATLAB code, it can include any number of commands or MATLAB software code wrapped around instrument commands to provide additional processing or analysis.

If the `set` code is defined as an instrument command, then the command written to the instrument will be the instrument command concatenated with a space and the value specified in the call to `set`. For example, the `set` code for the `DisplayContrast` property is defined as the instrument command `DISplay:CONTRast`. When the `set` function below is evaluated, the instrument command sent to the instrument will be `DISplay:CONTRast 54`.

```
set(obj,'DisplayContrast',54);
```

Get Code

The `get` code defines the code that is executed when the property value is queried with the `get` function. The `get` code can be defined as an instrument command that will be written to the instrument or it can be defined as MATLAB software code.

Note The code used for your property's `get` code and `set` code cannot include calls to the `fclose` or `fopen` functions on the interface object being used to access your instrument.

Accepted Property Values

You can define the values that the property can be set to so that only valid values are written to the instrument and an error would be returned before an invalid value could be written to the instrument.

- A property value can be defined as a double, a character vector, or a Boolean.
- A property value that is defined as a double can be restricted to accept only doubles within a certain range or a list of enumerated doubles. For example, a property could be defined to accept a double within the range of `[0 10]` or a property could be defined to accept one of the values `[1,7,8,10]`.
- A property value that is defined as a character vector can be restricted to accept a list of enumerated character vectors. For example, a property could be defined to accept the character vectors `min` and `max`.

Additionally, a property can be defined to accept multiple property value definitions. For example, a property could be defined to accept a double ranging between `[0 10]` or the character vectors `min` and `max`.

Property Value Dependencies

A property value can be dependent upon another property's value. For example, in controlling a power supply, the property `VoltageLevel` can be configured to the following values:

- A double ranging between 0 and 10 when the value of property `VoltageOutputRange` is `high`
- A double ranging between 0 and 5 when the value of property `VoltageOutputRange` is `low`

When `VoltageLevel` is configured, the value of `VoltageOutputRange` is queried. If the value of `VoltageOutputRange` is `high`, then `VoltageLevel` can be configured to a double ranging between 0 and 10. If the value of `VoltageOutputRange` is `low`, then `VoltageLevel` can be configured to a double ranging between 0 and 5.

Default Value

The default value of the property is the value that the property is configured to when the object is created.

Read-Only Value

The read-only value of the property defines when the property can be configured. Valid options are described below.

Read-Only Value	Description
Never	The property can be configured at all times with the `set` function.
While Open	The property can only be configured with the `set` function when the device object is not connected to the instrument. A device object is disconnected from the instrument with the `disconnect` function.
Always	The property cannot be configured with the `set` function.

Help Text

The help text provides information on the property. This information is returned with the `instrhelp` function.

```
instrhelp(obj,'PropertyName')
```

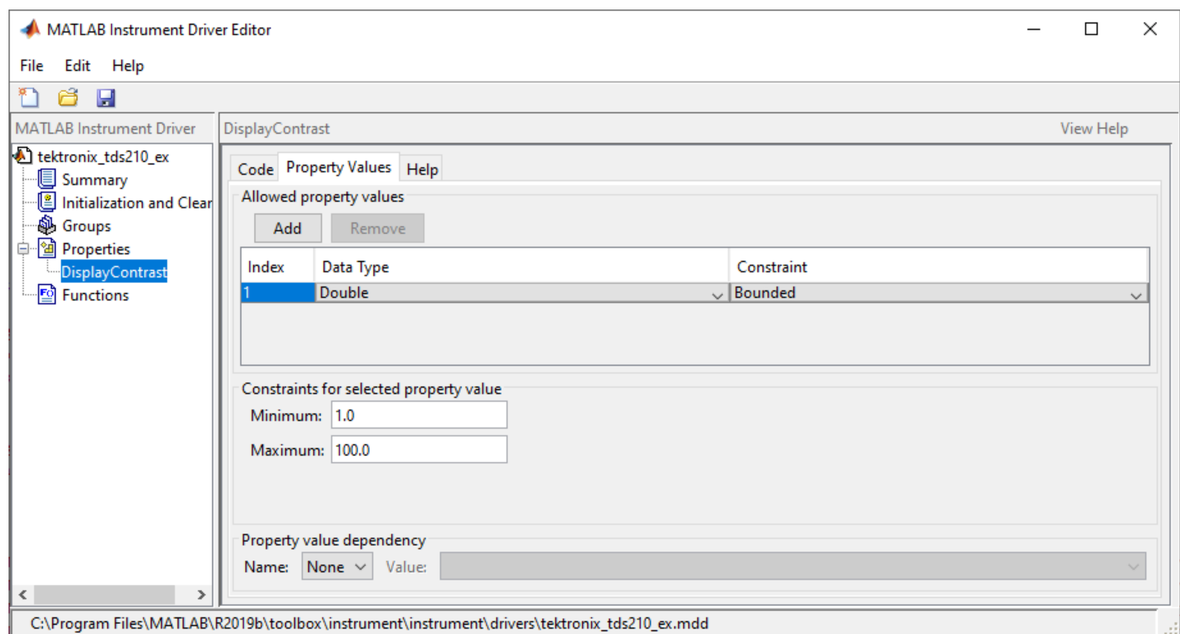
Examples of Properties

This section includes several examples of creating, setting, and reading properties, with steps for verifying the behavior of these properties.

Creating a Double-Bounded Property

This example creates a property that will configure the Tektronix TDS 210 oscilloscope's LCD display contrast. The oscilloscope display can be configured to a value in the range `[1 100]`. In the MATLAB instrument driver editor,

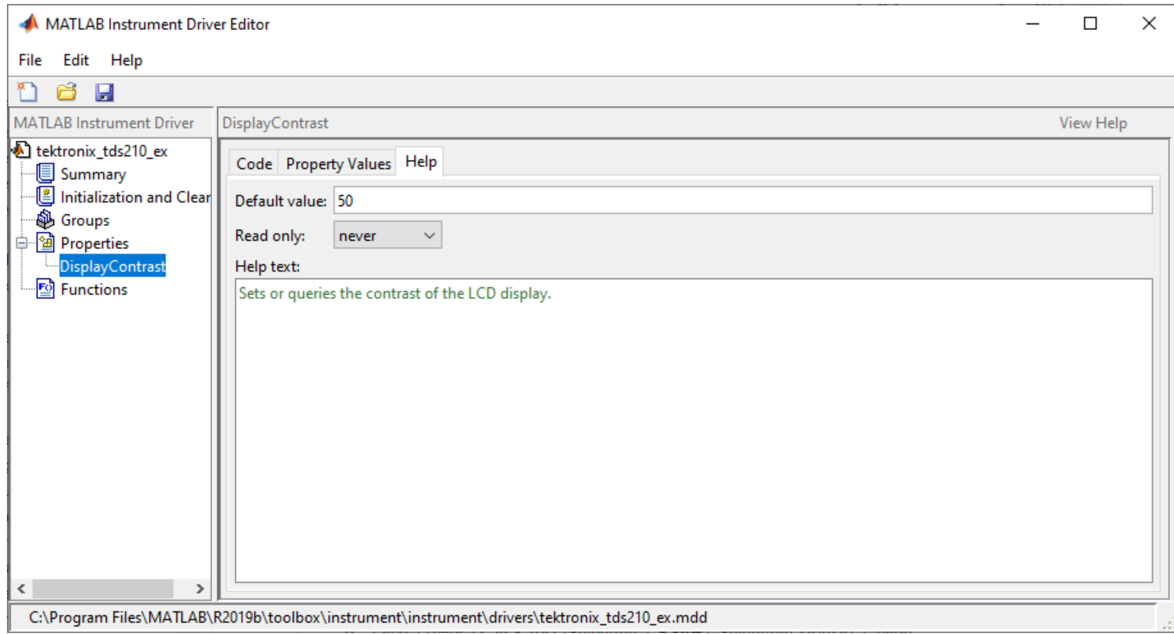
1. Select the `Properties` node in the tree.
2. Enter the property name, `DisplayContrast`, in the **Name** text field and click the **Add** button. The new property's name, `DisplayContrast`, appears in the **Property Name** table.
3. Expand the `Properties` node in the tree to display all the defined properties.
4. Select the `DisplayContrast` node from the properties displayed in the tree.
5. Select the **Code** tab to define the `set` and `get` commands for the `DisplayContrast` property.
   - Select `Instrument Commands` in the **Property style** field.
   - Enter `DISplay:CONTRast?` in the **Get command** text field.
   - Enter `DISplay:CONTRast` in the **Set command** text field.
6. Select the **Property Values** tab to define the allowed property values.
   - Select `Double` in the **Data Type** field.
   - Select `Bounded` in the **Constraint** field.
   - Enter `1.0` in the **Minimum** field.
   - Enter `100.0` in the **Maximum** field.



7. Select the **Help** tab to finish defining the property behavior.

- Enter `50` in the **Default value** text field.
- Select `never` in the **Read only** field.
- In the **Help text** field, enter `Sets or queries the contrast of the LCD display`.

8 Click the **Save** button.



Verifying the Behavior of the Property

This procedure verifies the behavior of the property. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2 View `DisplayContrast` property and its current value.

```
obj.DisplayContrast
```

```
ans =

    50
```

3 Calling `set` on the `DisplayContrast` property lists the values to which you can set the property.

```
set(obj,'DisplayContrast')
```

```
[ 1.0 to 100.0 ]
```

4 Try setting the property to values inside and outside of the specified range.

```
obj.DisplayContrast = 17;
obj.DisplayContrast
```

```
ans =

    17
```

```
obj.DisplayContrast = 120
```

```
Invalid value for DisplayContrast
Valid values: a value between 1.0 and 100.0.
```

5 View the help you wrote.

```
instrhelp(obj,'DisplayContrast')
```

```
   DISPLAYCONTRAST  [ 1.0 to 100.0 ]

   Sets or queries the contrast of the LCD display.
```

6 List the `DisplayContrast` characteristics that you defined in the **Property Values** and **Help** tabs.

```
info = propinfo(obj,'DisplayContrast')
```

```
info =

  struct with fields:

                Type: 'double'
          Constraint: 'bounded'
     ConstraintValue: [1 100]
        DefaultValue: 50
            ReadOnly: 'never'
   InterfaceSpecific: 1
```

7 Connect to your instrument to verify the `set` and `get` code.

```
connect(obj)
```

When you issue the `get` function in the MATLAB software, the `tektronix_tds210_ex.mdd` driver actually sends the `DISplay:CONTRast?` command to the instrument.

```
obj.DisplayContrast
```

```
ans =

    17
```

When you issue the `set` function in the MATLAB software, the `tektronix_tds210_ex.mdd` driver actually sends the `DISplay:CONTRast 34` command to the instrument.

```
obj.DisplayContrast = 34;
```

8 Finally, disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

Creating an Enumerated Property

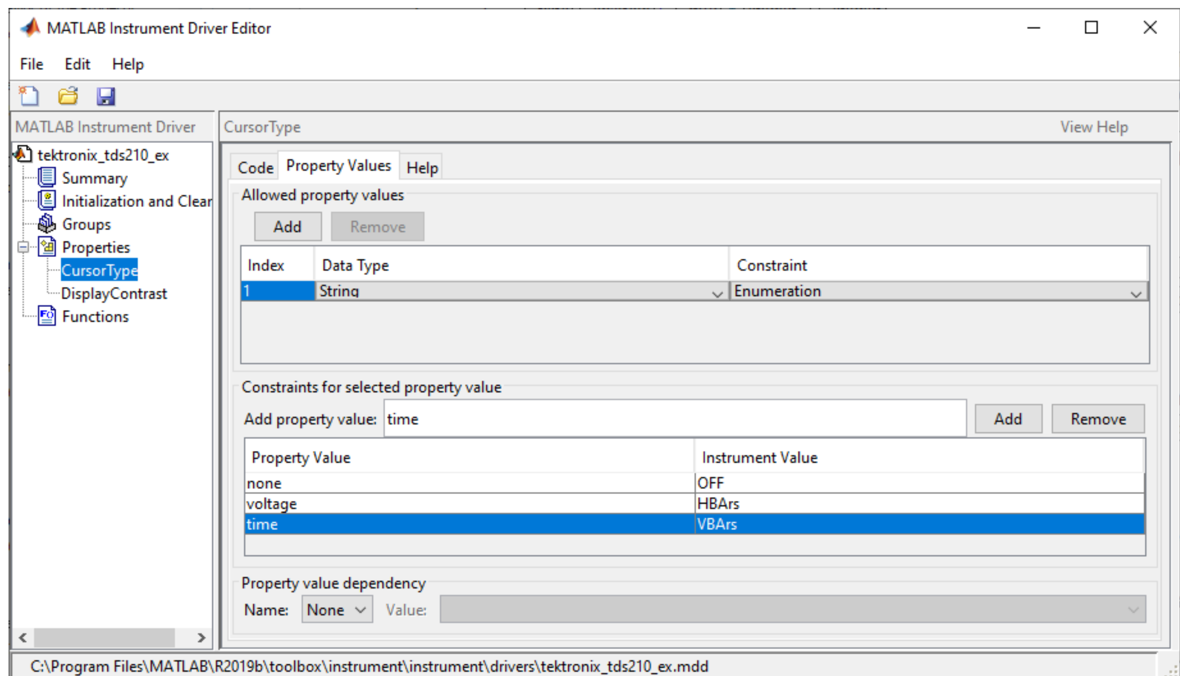
This example creates a property that will select and display the Tektronix TDS 210 oscilloscope's cursor. The oscilloscope allows two types of cursor. It supports a horizontal cursor that measures the

vertical units in volts, divisions, or decibels, and a vertical cursor that measures the horizontal units in time or frequency. In the MATLAB instrument driver editor,

1. Select the `Properties` node in the tree.
2. Enter the property name, `CursorType`, in the **Name** text field and click the **Add** button. The new property's name `CursorType` appears in the **Property Name** table.
3. Expand the `Properties` node to display all the defined properties.
4. Select the `CursorType` node from the properties displayed in the tree.
5. Select the **Code** tab to define the `set` and `get` commands for the `CursorType` property.

   - Select `Instrument Commands` in the **Property style** field.
   - Enter `CURSor:FUNCtion?` in the **Get Command** text field.
   - Enter `CURSor:FUNCtion` in the **Set Command** text field.
6. Select the **Property Values** tab to define the allowed property values.

   - Select `String` in the **Data Type** field.
   - Select `Enumeration` in the **Constraint** field.
   - Enter `none` in the **New property value** text field and click the **Add** button. Then enter `OFF` in the **Instrument Value** table field.
   - Similarly add the property value `voltage`, with instrument value `HBArs`.
   - Similarly add the property value `time`, with instrument value `VBArs`.



7. Select the **Help** tab to finish defining the property behavior.

   - Enter `none` in the **Default value** text field.
   - Select `never` in the **Read only** field.
   - In the **Help text** field, enter `Specifies the type of cursor`.
8. Click the **Save** button.

Verifying the Behavior of the Property

This procedure verifies the behavior of the property. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1. Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2. View the `CursorType` property's current value. Calling `get` on the object lists all its properties.

```
get(obj)

    ConfirmationFcn =
    DriverName = tektronix_tds210_ex.mdd
    DriverType = MATLAB interface object
    InstrumentModel =
    Interface = [1x1 gpib]
    LogicalName = GPIB0-2
    Name = scope-tektronix_tds210_ex
    ObjectVisibility = on
    RsrcName =
    Status = closed
    Tag =
    Timeout = 10
    Type = scope
    UserData = []

    SCOPE specific properties:
    CursorType = none
    DisplayContrast = 50
```

Calling `get` on the `CursorType` property lists its current value.

```
obj.CursorType


ans =

    'none'
```

3. View acceptable values for the `CursorType` property. Calling `set` on the object lists all its settable properties.

```
set(obj)

    ConfirmationFcn: string -or- function handle -or- cell array
    Name:
    ObjectVisibility: [ {on} | off ]
    Tag:
    Timeout:
    UserData:

    SCOPE specific properties:
    CursorType: [ {none} | voltage | time ]
    DisplayContrast: [ 1.0 to 100.0 ]
```

Calling `set` on the `CursorType` property lists the values to which you can set the property.

```
set(obj,'CursorType')

[ {none} | voltage | time ]
```

4 Try setting the property to valid and invalid values.

```
obj.CursorType = 'voltage';
obj.CursorType


ans =

    'voltage'

obj.CursorType = 'horizontal'

There is no enumerated value named 'horizontal'.
```

5 View the help you wrote.

```
instrhelp(obj,'CursorType')


   CURSORTYPE  [ {none} | voltage | time ]

   Specifies the type of cursor.
```

6 List the `CursorType` characteristics that you defined in the **Property Values** and **Help** tabs.

```
info = propinfo(obj,'CursorType')


info = 

  struct with fields:

                 Type: 'string'
           Constraint: 'enum'
      ConstraintValue: {3×1 cell}
         DefaultValue: 'none'
             ReadOnly: 'never'
    InterfaceSpecific: 1

info.ConstraintValue


ans =

  3×1 cell array

    {'none'   }
    {'voltage'}
    {'time'   }
```

7 Connect to your instrument to verify the `set` and `get` code.

```
connect(obj)
```

When you issue the `set` function in the MATLAB software, the `tektronix_tds210_ex.mdd` driver actually sends the `CURSor:FUNCtion VBArs` command to the instrument.

```
obj.CursorType = 'time';
```

When you issue the `get` function in the MATLAB software, the `tektronix_tds210_ex.mdd` driver actually sends the `CURSor:FUNCtion?` command to the instrument.

```
obj.CursorType
```

```
ans =

    'time'
```

8 Finally disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

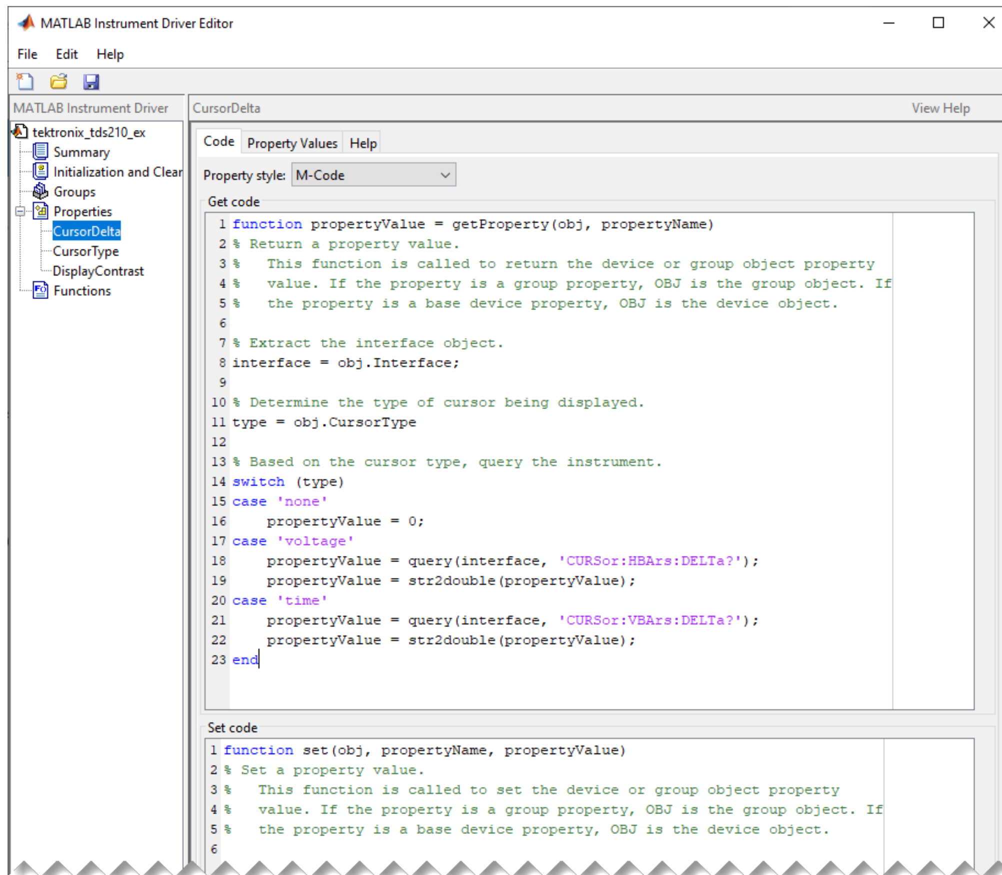
A MATLAB Code Style Property

This example creates a property that will return the difference between two cursors of the Tektronix TDS 210 oscilloscope. The oscilloscope allows two types of cursor. It supports a horizontal cursor that measures the vertical units in volts, divisions, or decibels, and a vertical cursor that measures the horizontal units in time or frequency. The previous example created a property, `CursorType`, that selects and displays the oscilloscope's cursor. In the MATLAB instrument driver editor,

1 Select the `Properties` node in the tree.
2 Enter the property name, `CursorDelta`, in the **New Property** text field and click **Add**. The new property's name, `CursorDelta`, appears in the **Property Name** table.
3 Expand the `Properties` node to display all the defined properties.
4 Select the `CursorDelta` node from the properties displayed in the tree.
5 Select the **Code** tab to define the set and get commands for the `CursorDelta` property.

- Select `M-Code` in the **Property style** field.
- Since the `CursorDelta` property is read-only, no MATLAB software code will be added to the **Set code** text field.
- The following MATLAB software code is added to the **Get code** text field.

```
% Extract the interface object.
interface = obj.Interface;

% Determine the type of cursor being displayed.
type = obj.CursorType

% Based on the cursor type, query the instrument.
switch (type)
case 'none'
    propertyValue = 0;
case 'voltage'
    propertyValue = query(interface, 'CURSor:HBArs:DELTa?');
    propertyValue = str2double(propertyValue);
case 'time'
    propertyValue = query(interface, 'CURSor:VBArs:DELTa?');
    propertyValue = str2double(propertyValue);
end
```



6 Select the **Property Values** tab to define the allowed property values.

- Select `Double` in the **Data Type** field.
- Select `None` in the **Constraint** field.

7 Select the **Help** tab to finish defining the property behavior.

- Enter `0` in the **Default value** text field.
- Select `always` in the **Read only** field.
- In the **Help text** field, enter `Returns the difference between the two cursors`.

8 Click the **Save** button.

Verifying the Behavior of the Property

This procedure verifies the behavior of the property. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2 View the `CursorDelta` property's current value. Calling `get` on the object lists all its properties.

```
get(obj)
```

```
    ConfirmationFcn =
    DriverName = tektronix_tds210_ex.mdd
    DriverType = MATLAB interface object
    InstrumentModel =
    Interface = [1x1 gpib]
    LogicalName = GPIB0-2
    Name = scope-tektronix_tds210_ex
    ObjectVisibility = on
    RsrcName =
    Status = closed
    Tag =
    Timeout = 10
    Type = scope
    UserData = []

    SCOPE specific properties:
    CursorDelta = 0
    CursorType = none
    DisplayContrast = 50
```

3 View the `CursorDelta` property's current value.

```
obj.CursorDelta
```

```
ans =

     0
```

4 Calling `set` on the object lists all its settable properties. Note that as a read-only property, `CursorDelta` is not listed in the output.

```
set(obj)
```

```
    ConfirmationFcn: string -or- function handle -or- cell array
    Name:
    ObjectVisibility: [ {on} | off ]
    Tag:
    Timeout:
    UserData:

    SCOPE specific properties:
    CursorType: [ {none} | voltage | time ]
    DisplayContrast: [ 1.0 to 100.0 ]
```

5 Setting the property to a value results in an error message.

```
obj.CursorDelta = 4;
```

```
Changing the 'CursorDelta' property of device objects is not allowed.
```

6 View the help you wrote.

```
instrhelp(obj,'CursorDelta')
```

```
   CURSORDELTA  (double)  (read only)

   Returns the difference between the two cursors.
```

7 List the `CursorDelta` characteristics that you defined in the **Property Values** and **Help** tabs.

```
info = propinfo(obj,'CursorDelta')
```

```
info = 

  struct with fields:

                 Type: 'double'
           Constraint: 'none'
      ConstraintValue: []
         DefaultValue: 0
             ReadOnly: 'always'
    InterfaceSpecific: 1
```

8 Connect to your instrument to verify the `get` code.

```
connect(obj)
```

When you issue the `get` function in the MATLAB software, the `tektronix_tds210_ex.mdd` driver actually executes the MATLAB software code that was specified.

```
obj.CursorDelta
```

```
ans =

     1.6000
```

9 Finally, disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

Functions

In this section...
"Understanding Functions" on page 19-25
"Function Components" on page 19-25
"Examples of Functions" on page 19-26

Understanding Functions

Functions allow you to call the instrument to perform some task or tasks, which may return results as text data or numeric data. The function may involve a single command to the instrument, or a sequence of instrument commands. A function may include the MATLAB software code to determine what commands are sent to the instrument or to perform analysis on data returned from the instrument. For example, a function may request that a meter run its self-calibration, returning the status as a result. Another function may read a meter's scaling, request a measurement, adjust the measured data according to the scale reading, and then return the result.

Function Components

The behavior of the function is defined by the components described below.

MATLAB Code

The MATLAB code defines the code that is executed when the function is evaluated with the `invoke` function. The code can be defined as an instrument command that will be written to the instrument or it can be defined as the MATLAB software code.

If the code is defined as an instrument command, the instrument command can be defined to take an input argument. All occurrences of `<input argument name>` in the instrument command are substituted with the input value passed to the `invoke` function. For example, if a function is defined with an input argument, `start`, and the instrument command is defined as `Data:Start <start>`, and a start value of `10` is passed to the `invoke` function, the command `Data:Start 10` is written to the instrument.

If the code is defined as an instrument command, the instrument command can also be defined to return an output argument. The output argument can be returned as numeric data or as text data.

If the code is defined as the MATLAB software code, you can determine which commands are sent to the instrument, and the data results from the instrument can be manipulated, adjusted, or analyzed as needed.

Note The code used for your function's MATLAB software code cannot include calls to the `fclose` or `fopen` functions on the interface object being used to access your instrument.

Help Text

The help text provides information on the function.

Examples of Functions

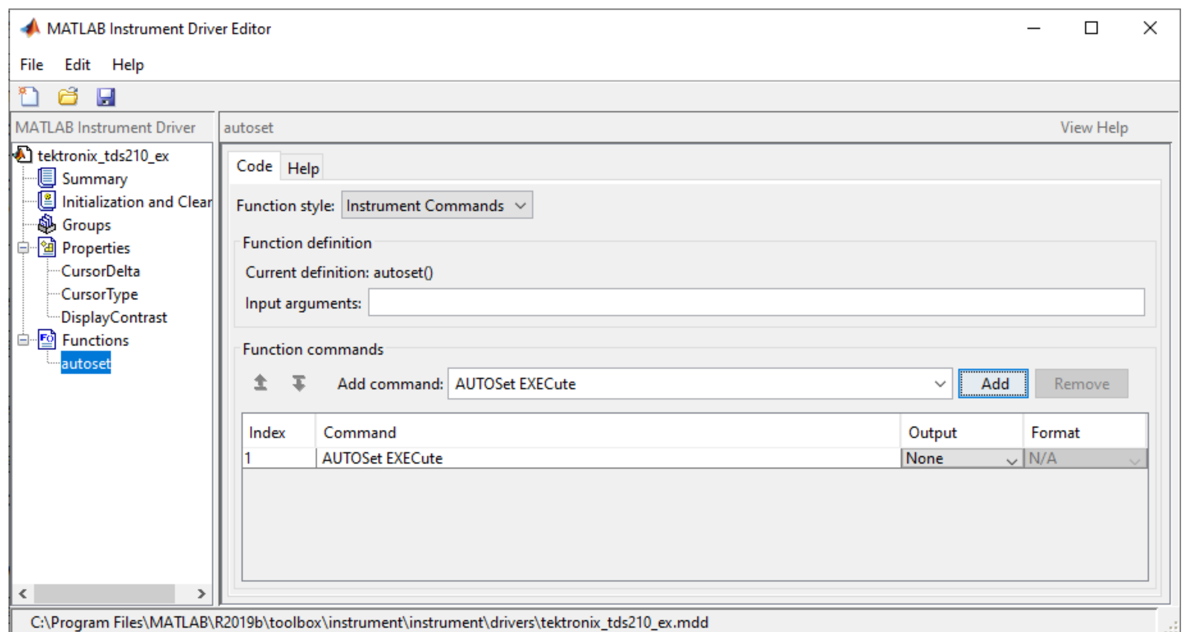
This section includes several examples of functions, and steps to verify the behavior of these functions.

Simple Function

This example creates a function that will cause the Tektronix TDS 210 oscilloscope to adjust its vertical, horizontal and trigger controls to display a stable waveform. In the MATLAB instrument driver editor,

1. Select the `Functions` node in the tree.
2. Enter the function name, `autoset`, in the **Add function** text field and click the **Add** button. The new function's name, `autoset`, appears in the **Function Name** table.
3. Expand the `Functions` node to display all the defined functions.
4. Select the `autoset` node from the functions displayed in the tree.
5. Select the **Code** tab to define commands executed for this function.

   - Select `Instrument Commands` in the **Function style** field.
   - In the **Function commands** pane, enter `AUTOSet EXECute` in the **Add command** field and click the **Add** button.



6. Select the **Help** tab to define the help text for this function.

   - In the **Help text** field, enter `INVOKE(OBJ, 'autoset') causes the oscilloscope to adjust its vertical, horizontal, and trigger controls to display a stable waveform`.
7. Click the **Save** button.

Verifying the Behavior of the Function

This procedure verifies the behavior of the function. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary

address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1. Create the device object, `obj`, using the `icdevice` function.

   ```
   g = gpib('mcc',0,2);
   obj = icdevice('tektronix_tds210_ex.mdd',g);
   ```
2. View the method you created.

   ```
   methods(obj)


   Methods for class icdevice:

   class          display        icdevice       instrnotify    methods        size
   connect        end            inspect        instrument     ne             subsasgn
   ctranspose     eq             instrcallback  invoke         obj2mfile      subsref
   delete         fieldnames     instrfind      isa            openvar        vertcat
   devicereset    get            instrfindall   isequal        propinfo
   disconnect     geterror       instrhelp      isvalid        selftest
   disp           horzcat        instrhwinfo    length         set

   Driver specific methods for class icdevice:

   autoset
   ```
3. View the help you wrote.

   ```
   instrhelp(obj,'autoset')


      INVOKE(OBJ, 'autoset') causes the oscilloscope to adjust its vertical,
      horizontal, and trigger controls to display a stable waveform.
   ```
4. Using the controls on the instrument, set the scope so that its display is unstable. For example, set the trigger level outside the waveform range so that the waveform scrolls across the display.
5. Connect to your instrument and execute the function. Observe how the display of the waveform stabilizes.

   ```
   connect(obj)
   invoke(obj,'autoset')
   ```
6. Disconnect from your instrument and delete the object.

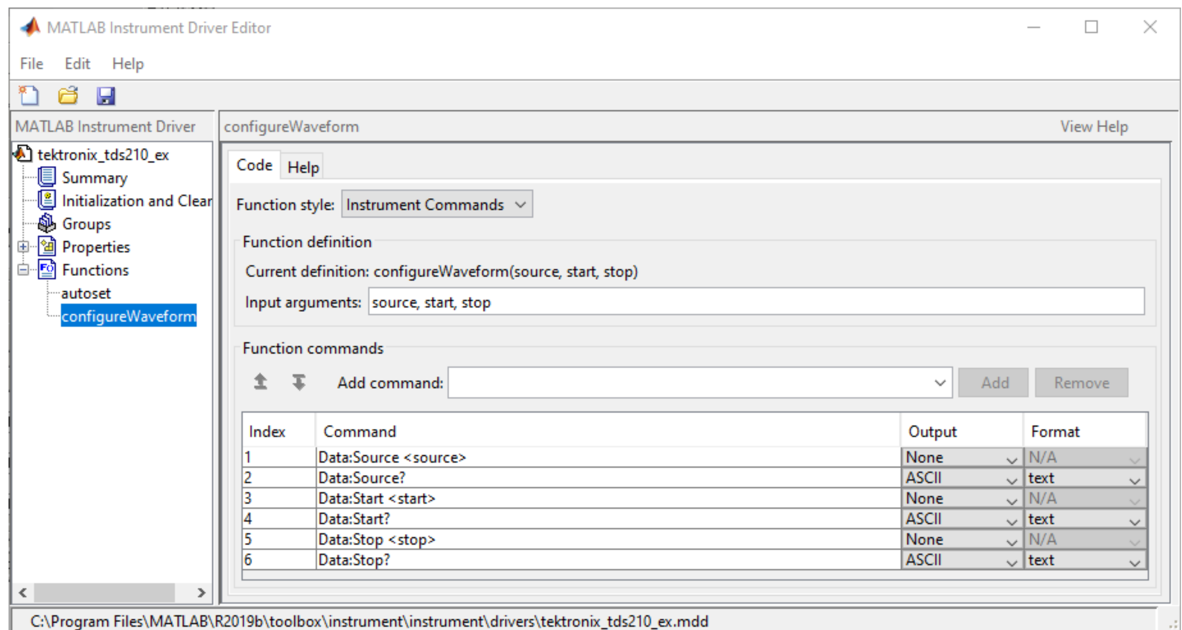
   ```
   disconnect(obj)
   delete([obj g])
   ```

Function with Instrument Commands that Use Input and Output Arguments

This example creates a function that configures which waveform will be transferred from the Tektronix TDS 210 oscilloscope, and configures the waveform's starting and ending data points. In the MATLAB instrument driver editor,

1. Select the `Functions` node in the tree.
2. Enter the function name, `configureWaveform`, in the **Add function** text field and click the **Add** button. The new function's name, `configureWaveform`, appears in the **Function Name** pane.
3. Expand the `Functions` node to display all the defined functions.
4. Select the `configureWaveform` node from the functions displayed in the tree.
5. Select the **Code** tab to define commands executed for this function.

- Select `Instrument Commands` in the **Function style** field.
- Enter the input arguments `source, start, stop` in the **Input arguments** field.
- Enter `Data:Source <source>` in the **Add command** field and click the **Add** button. In the table, select an **Output** type of `None` and a **Format** type of `N/A`.
- Similarly, add the command: `Data:Source?` with `ASCII` **Output** and `text` **Format**.
- Similarly, add the command: `Data:Start <start>` with `NONE` **Output** and `N/A` **Format**.
- Similarly, add the command: `Data:Start?` with `ASCII` **Output** and `numeric` **Format**.
- Similarly, add the command: `Data:Stop <stop>` with `NONE` **Output** and `N/A` **Format**.
- Similarly, add the command: `Data:Stop?` with `ASCII` **Output** and `numeric` **Format**.



6 Select the **Help** tab to define the help text for this function.

- In the **Help text** field, enter `[SOURCE, START, STOP] = INVOKE(OBJ, 'configureWaveform', SOURCE, START, STOP) configures the waveform that will be transferred from the oscilloscope.`

7 Click the **Save** button.

Verifying the Behavior of the Function

This procedure verifies the behavior of the function. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2 View the method you created.

```
methods(obj)
```

```
Methods for class icdevice:

class               fieldnames          instrhwinfo         obj2mfile
connect             get                 instrnotify         openvar
ctranspose          geterror            instrument          propinfo
delete              horzcat             invoke              selftest
devicereset         icdevice            isa                 set
disconnect          inspect             isequal             size
disp                instrcallback       isvalid             subsasgn
display             instrfind           length              subsref
end                 instrfindall        methods             vertcat
eq                  instrhelp           ne

Driver specific methods for class icdevice:

autoset             configureWaveform
```

3 View the help you wrote.

```
instrhelp(obj,'configureWaveform')
```

```
   [SOURCE, START, STOP] = INVOKE(OBJ, 'configureWaveform', SOURCE, START,
   STOP) configures the waveform that will be transferred from the oscilloscope.
```

4 Connect to your instrument and execute the function.

```
connect(obj)
[source,start,stop] = invoke(obj,'configureWaveform','CH1',1,500)
```

```
source =

    'CH1'


start =

     1


stop =

        500
```

```
[source,start,stop] = invoke(obj,'configureWaveform','CH2',0,3500)
```

```
source =

    'CH2'


start =

     1


stop =
```

```
2500
```

5 Disconnect from your instrument and delete the object.

```
disconnect(obj)
delete([obj g])
```

MATLAB Code Style Function

This example creates a function that will transfer and scale the waveform from the Tektronix TDS 210 oscilloscope. In the MATLAB instrument driver editor,

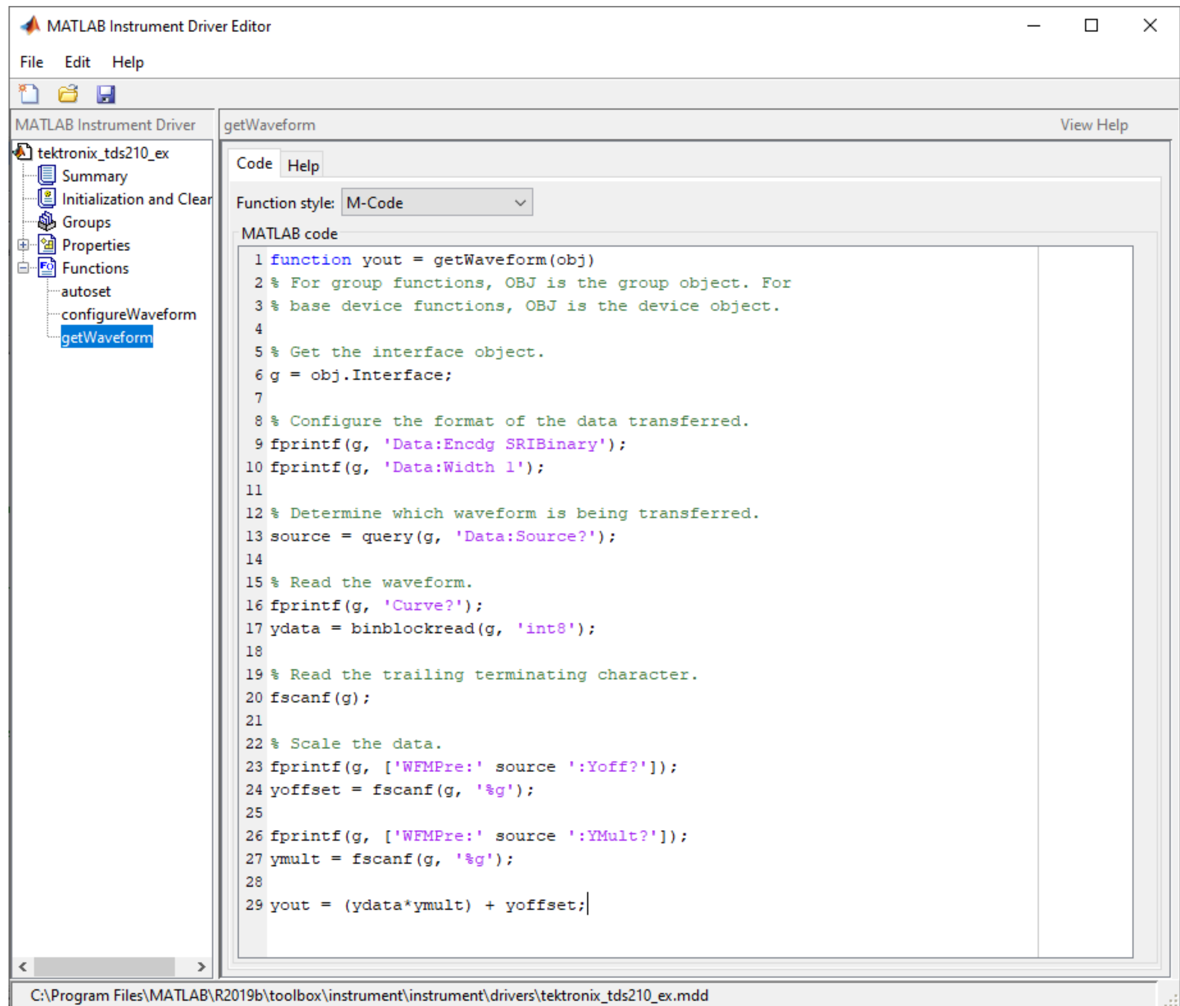
1 Select the `Functions` node in the tree.
2 Enter the function name, `getWaveform`, in the **Add function** text field and click the **Add** button. The new function's name, `getWaveform`, appears in the **Function Name** table.
3 Expand the `Functions` node to display all the defined functions.
4 Select the `getWaveform` node from the functions displayed in the tree.
5 Select the **Code** tab to define commands executed for this function.

- Select `M-Code` in the **Function style** field.
- Update the function line in the **Define MATLAB function** text field to include an output argument.

```
function yout = getWaveform(obj)
```

- Add the following MATLAB software code to the **Define MATLAB function** text field. (The instrument may require a short pause before any statements that read a waveform, to allow its completion of the data collection.)

```
% Get the interface object.
g = obj.Interface;

% Configure the format of the data transferred.
fprintf(g, 'Data:Encdg SRIBinary');
fprintf(g, 'Data:Width 1');

% Determine which waveform is being transferred.
source = query(g, 'Data:Source?');

% Read the waveform.
fprintf(g, 'Curve?');
ydata = binblockread(g, 'int8');

% Read the trailing terminating character.
fscanf(g);

% Scale the data.
fprintf(g, ['WFMPre:' source ':Yoff?']);
yoffset = fscanf(g, '%g');

fprintf(g, ['WFMPre:' source ':YMult?']);
ymult = fscanf(g, '%g');

yout = (ydata*ymult) + yoffset;
```



6 Click the **Help** tab to define the help text for this function.

- In the **Help text** field, enter `DATA = INVOKE(OBJ, 'getWaveform') transfers and scales the waveform from the oscilloscope`.

7 Click the **Save** button.

Verifying the Behavior of the Function

This procedure verifies the behavior of the function. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2 View the method you created.

```
methods(obj)
```

```
Methods for class icdevice:
```

```
class               fieldnames          instrhwinfo         obj2mfile
connect             get                 instrnotify         openvar
ctranspose          geterror            instrument          propinfo
delete              horzcat             invoke              selftest
devicereset         icdevice            isa                 set
disconnect          inspect             isequal             size
disp                instrcallback       isvalid             subsasgn
display             instrfind           length              subsref
end                 instrfindall        methods             vertcat
eq                  instrhelp           ne

Driver specific methods for class icdevice:

autoset             configureWaveform  getWaveform
```

3 View the help you wrote.

```
instrhelp(obj,'getWaveform')
```

```
   DATA = INVOKE(OBJ, 'getWaveform') transfers and scales the waveform from
   the oscilloscope.
```

4 Connect to your instrument and execute the function.

```
connect(obj)
```

Configure the waveform that is going to be transferred.

```
invoke(obj,'configureWaveform','CH1',1,500);
```

Transfer the waveform.

```
data = invoke(obj,'getWaveform');
```

Analyze and view the waveform.

```
size(data)
```

```
ans =

   500     1
```

```
plot(data)
```

5 Disconnect from your instrument and delete the object.

```
disconnect(obj)
delete([obj g])
```

Groups

In this section…
“Group Components” on page 19-33
“Examples of Groups” on page 19-33

Group Components

A group may be used to set or query the same property on several elements, or to query several related properties, at the same time. For example, all input channels on an oscilloscope can be scaled to the same value with a single command; or all current measurement setups can be retrieved and viewed at the same time.

A group consists of one or more group objects. The objects in the group share a set of properties and functions. Using these properties and functions you can control the features of the instrument represented by the group. In order for the group objects to control the instrument correctly, the group must define a selection command for the group and an identification string for each object in the group.

Selection Command

The selection command is an instrument command that configures the instrument to use the capability or physical component represented by the current group object. Note, the instrument might not have a selection command.

Identification String

The identification string identifies an object in the group. The number of identification strings listed by the group defines the number of objects in the group. The identification string can be substituted into the instrument commands written to the instrument.

When a group object instrument command is written to the instrument, the following steps occur:

1. The selection command for the group is determined by the driver.
2. The identification string for the group object is determined by the driver.
3. If the selection command contains the string `<ID>`, it is replaced with the identification string.
4. The selection command is written to the instrument. If empty, nothing is written to the instrument.
5. If the instrument command contains the string `<ID>`, it is replaced with the identification string.
6. The instrument command is written to the instrument.

Examples of Groups

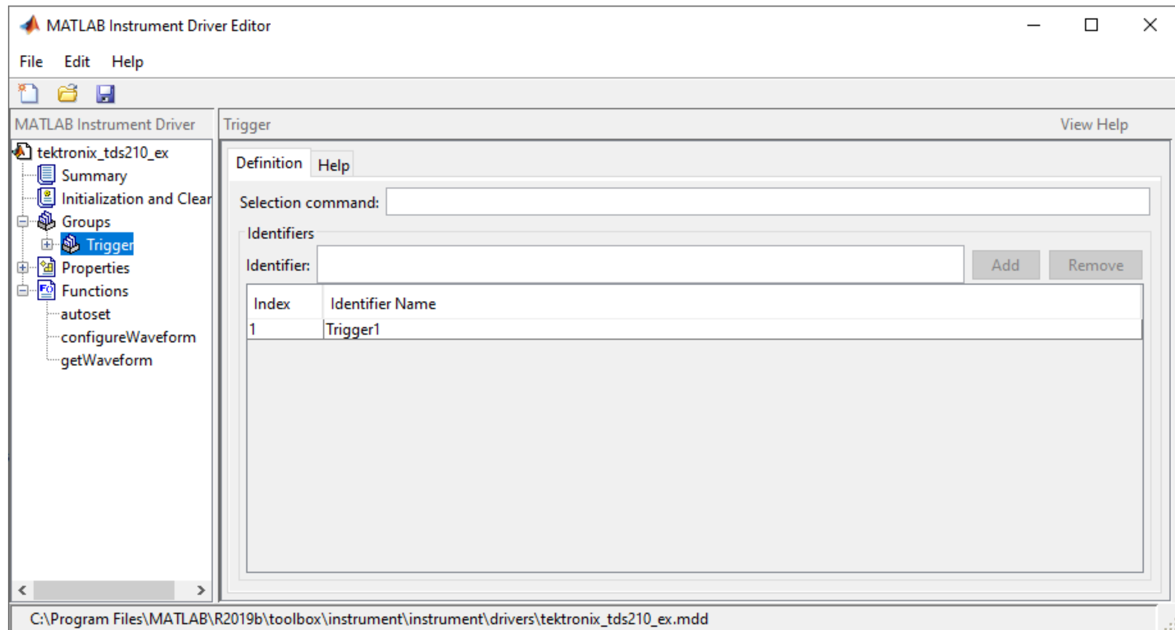
This section includes several examples of groups, with steps to verify the code.

Creating a One-Element Group

This example combines the trigger capabilities of the Tektronix TDS 210 oscilloscope into a trigger group. The oscilloscope allows the trigger source and slope settings to be configured. In the MATLAB instrument driver editor,

1 Select the `Groups` node in the tree.
2 Enter the group name, `Trigger`, in the **Add Group** text field and click **Add**.
3 Expand the `Groups` node to display all the defined groups.
4 Select the `Trigger` node in the tree.
5 Select the **Definition** tab.

Since the oscilloscope has only one trigger, there is not a command that will select the current trigger. The **Selection command** text field will remain empty.



6 Select the **Help** tab to finish defining the group behavior.

In the **Help text** field, enter `Trigger is a trigger group. The trigger group object contains properties that configure and query the oscilloscope's triggering capabilities.`
7 Click the **Save** button.

Verifying the Group Behavior

This procedure verifies the group information defined. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```
2 View the group you created. Note that the `HwName` property is the group object identification string.

```
obj.Trigger
```

```
   HwIndex:    HwName:     Type:            Name:
   1           Trigger1    scope-trigger    Trigger1
```

3 View the help.

```
instrhelp(obj, 'Trigger')
```

```
TRIGGER

Trigger is a trigger group. The trigger group object contains properties
that configure and query the oscilloscope's triggering capabilities.
```

4 Delete the objects.

```
delete([obj g])
```

Defining the Group Object Properties for a One-Element Group

This example defines the properties for the `Trigger` group object created in the previous example. The Tektronix TDS 210 oscilloscope can trigger from CH1 or CH2 when the data has a rising or falling slope.

First, the properties `Source` and `Slope` are added to the trigger group object. In the MATLAB instrument driver editor,

1. Expand the `Trigger` group node to display the group object's properties and functions.
2. Select the `Properties` node to define the `Trigger` group object properties.
3. Enter the property name `Source` in the **Add property** text field and click the **Add** button
4. Enter the property name `Slope` in the **Add property** text field and click the **Add** button.
5. Expand the `Properties` node to display the group object's properties.

Next, define the behavior of the `Source` property:

1. Select the `Source` node in the tree.
2. Select the **Code** tab to define the `set` and `get` commands for the `Source` property.
   - Select `Instrument Commands` in the **Property style** field.
   - Enter `TRIGger:MAIn:EDGE:SOUrce?` in the **Get command** text field.
   - Enter `TRIGger:MAIn:EDGE:SOUrce` in the **Set command** text field.
3. Select the **Property Values** tab to define the allowed property values.
   - Select `String` in the **Data Type** field.
   - Select `Enumeration` in the **Constraint** field.
   - Enter `CH1` in the **Add property value** text field and click the **Add** button. Then enter `CH1` in the **Instrument Value** table field.
   - Similarly, add the enumeration: `CH2`, `CH2`.
4. Select the **Help** tab to finish defining the property behavior.
   - Enter `CH1` in the **Default value** text field.
   - Select `never` in the **Read only** field.
   - In the **Help text** field, enter `Specifies the source for the main edge trigger`.

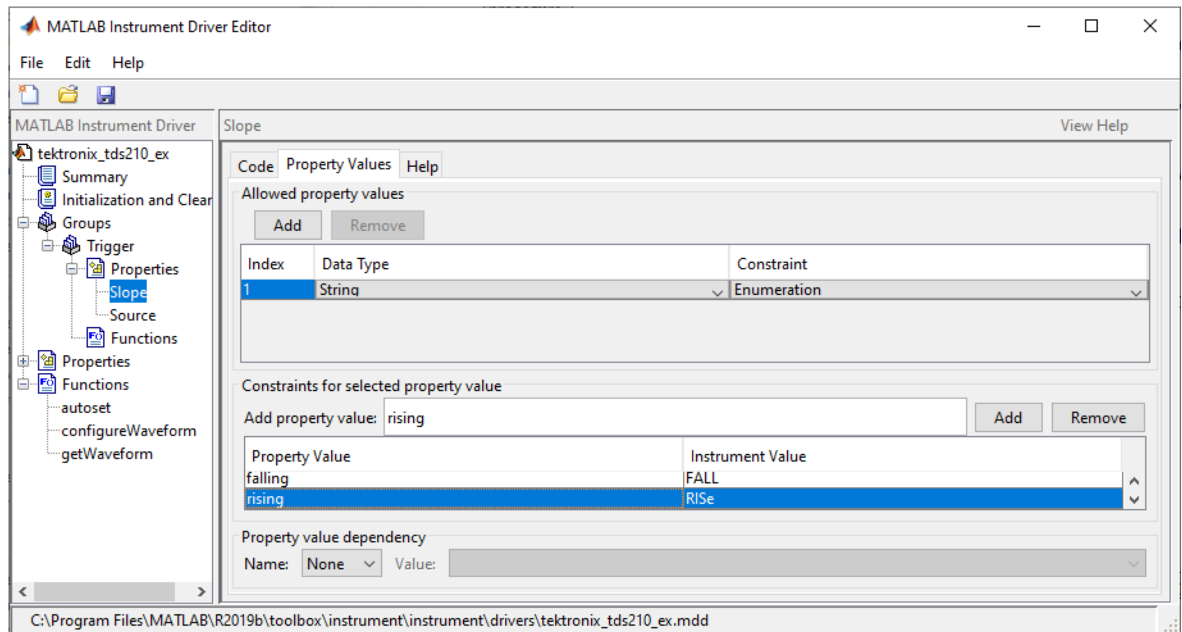
Next, define the behavior of the `Slope` property:

1. Select the `Slope` node in the tree.
2. Select the **Code** tab to define the set and get commands for the `Slope` property.

- Select `Instrument Commands` in the **Property style** field.
- Enter `TRIGger:MAIn:EDGE:SLOpe?` in the **Get command** text field.
- Enter `TRIGger:MAIn:EDGE:SLOpe` in the **Set command** text field.

3 Select the **Property Values** tab to define the allowed property values.

- Select `String` in the **Data Type** field.
- Select `Enumeration` in the **Constraint** field.
- Enter `falling` in the **Add property value** text field and click the **Add** button. Then enter `FALL` in the **Instrument Value** table field.
- Similarly add the enumeration: `rising`, `RISe`.



4 Select the **Help** tab to finish defining the property behavior.

- Enter `falling` in the **Default value** text field.
- Select `never` in the **Read only** field.
- In the **Help text** field, enter `Specifies a rising or falling slope for the main edge trigger`.

5 Click the **Save** button.

Verifying Properties of the Group Object in MATLAB

This procedure verifies the properties of the `Trigger` group object. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2 Extract the trigger group objects, `t`, from the device object.

```
t = obj.Trigger
```

```
   HwIndex:    HwName:     Type:             Name:
   1           Trigger1    scope-trigger     Trigger1
```

3 Access specific properties to list its current value.

```
t.Source
```

```
ans =

    'CH1'
```

```
t.Slope
```

```
ans =

    'falling'
```

4 Calling `set` on a specific property lists the values to which you can set the property.

```
set(t,'Source')
```

```
[ {CH1} | CH2 ]
```

```
set(t,'Slope')
```

```
[ {falling} | rising ]
```

5 Try setting the property to valid and invalid values.

```
t.Source = 'CH2';
t.Slope = 'rising';
```

```
t.Source
```

```
ans =

    'CH2'
```

```
t.Slope
```

```
ans =

    'rising'
```

```
t.Source = 'CH3'
```

```
There is no enumerated value named 'CH3'.
```

```
t.Slope = 'steady'
```

```
There is no enumerated value named 'steady'.
```

6 View the help you wrote.

```
instrhelp(t,'Source')
```

```
   SOURCE  [ {CH1} | CH2 ]
```

```
   Specifies the source for the main edge trigger.
```

```
instrhelp(t,'Slope')
```

```
   SLOPE  [ {falling} | rising ]

   Specifies a rising or falling slope for the main edge trigger.
```

7 List the group object characteristics that you defined in the **Property Values** and **Help** tabs.

```
propinfo(t,'Source')
```

```
ans = 

  struct with fields:

                Type: 'string'
          Constraint: 'enum'
     ConstraintValue: {2×1 cell}
        DefaultValue: 'CH1'
            ReadOnly: 'never'
   InterfaceSpecific: 1
```

```
propinfo(t,'Slope')
```

```
ans = 

  struct with fields:

                Type: 'string'
          Constraint: 'enum'
     ConstraintValue: {2×1 cell}
        DefaultValue: 'falling'
            ReadOnly: 'never'
   InterfaceSpecific: 1
```

8 Connect to your instrument to verify the `set` and `get` code.

```
connect(obj)
```

Note When you issue the `get` function on the `Source` property for the trigger object, the `textronix_tds210_ex.mdd` driver actually sends the `TRIGger:MAIn:EDGE:SOUrce?` command to the instrument.

```
t.Source
```

```
ans =

    'CH1'
```

Note When you issue the `set` function on the `Slope` property for the trigger object, the `textronix_tds210_ex.mdd` driver actually sends the `TRIGger:MAIn:EDGE:SLOpe RISe` command to the instrument.

```
t.Slope = 'rising';
```

9 Disconnect from your instrument and delete the objects.

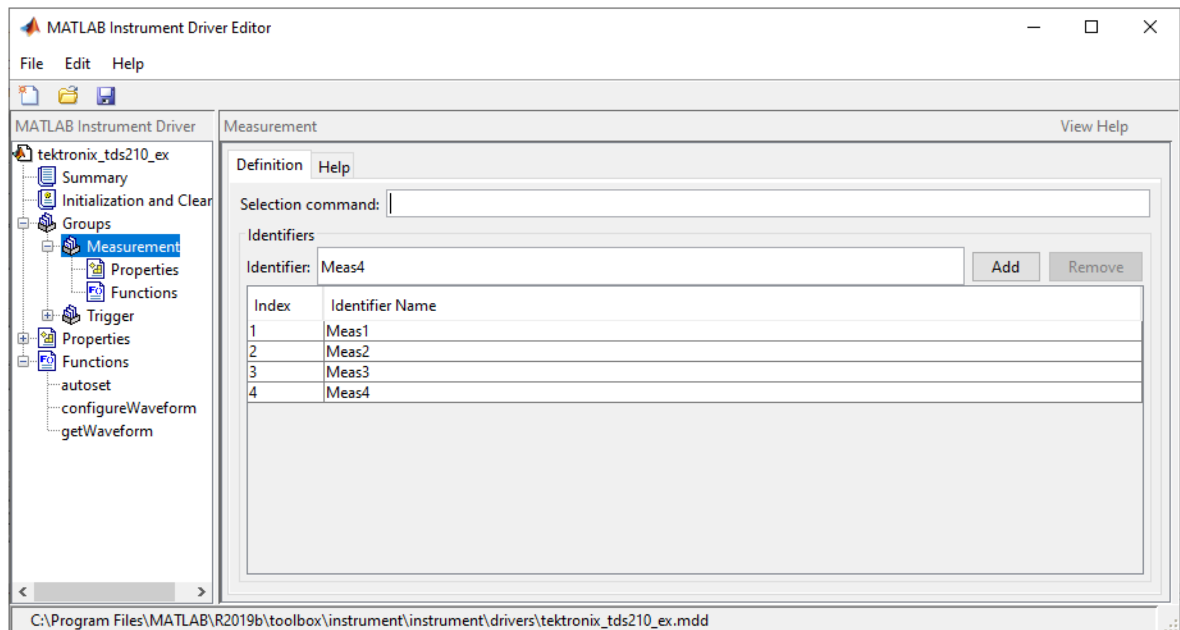
```
disconnect(obj)
delete([obj g])
```

Creating a Four-Element Group

This example combines the measurement capabilities of the Tektronix TDS 210 oscilloscope into a measurement group. The oscilloscope allows four measurements to be taken at a time. In the MATLAB instrument driver editor,

1 Select the `Groups` node in the tree.
2 Enter the group name, `Measurement`, in the **Add group** text field and click **Add**.
3 Expand the `Groups` node to display all the defined groups.
4 Select the `Measurement` node in the tree.
5 Select the **Definition** tab.

- The oscilloscope does not define an instrument command that will define the measurement that is currently being calculated. The **Selection command** text field will remain empty.
- In the **Identifier Name** listing, change `Measurement1` to `Meas1` to define the identification string for the first measurement group object in the group.
- Enter the identifiers `Meas2`, `Meas3`, and `Meas4` for the remaining measurement group objects by typing each in the **Identifier** text field and clicking **Add** after each.



6 Select the **Help** tab to finish defining the group behavior.

- In the **Help text** field, enter `Measurement is an array of measurement group objects. A measurement group object contains properties related to each supported measurement on the oscilloscope.`

7 Click the **Save** button.

Verifying the Group Behavior

This procedure verifies the group information defined. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1. Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2. View the group you created. Note that the `HwName` property is the group object `get(obj)`.

```
obj.Measurement
```

```
   HwIndex:    HwName:    Type:                 Name:
   1           Meas1      scope-measurement     Measurement1
   2           Meas2      scope-measurement     Measurement2
   3           Meas3      scope-measurement     Measurement3
   4           Meas4      scope-measurement     Measurement4
```

3. View the help.

```
instrhelp(obj,'Measurement')
```

```
   MEASUREMENT

   Measurement is an array of measurement group objects. A measurement group
   object contains properties related to each supported measurement on the
   oscilloscope.
```

4. Delete the objects.

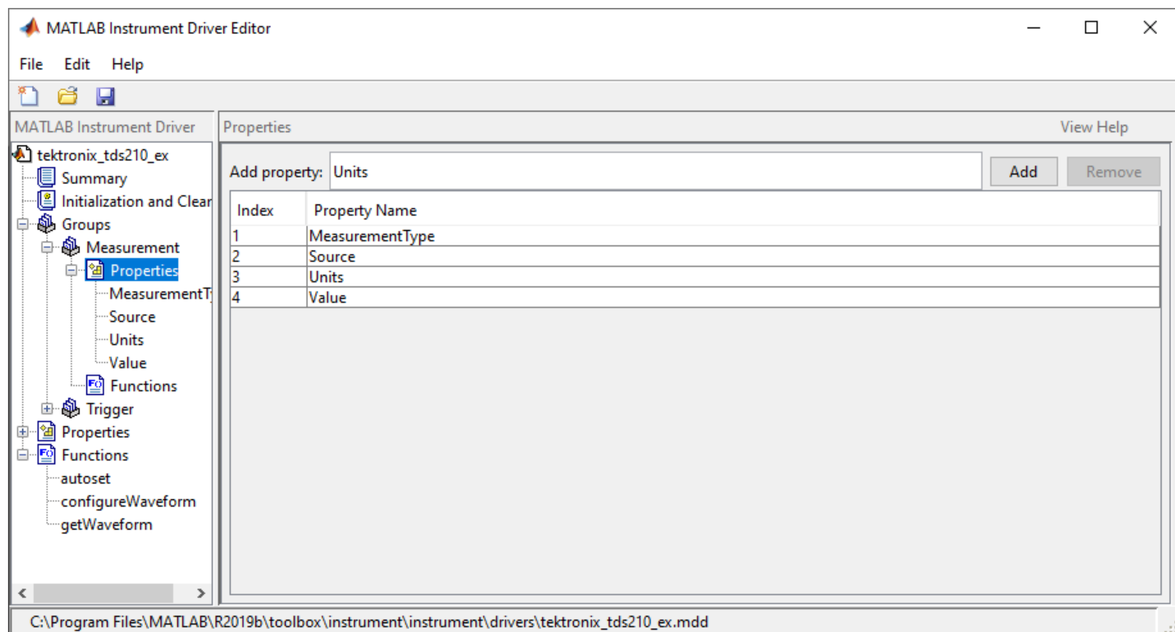
```
delete([obj g])
```

Defining the Group Object Properties for a Four-Element Group

This example defines the properties for the `Measurement` group object created in the previous example. The Tektronix TDS 210 oscilloscope can calculate the frequency, mean, period, peak to peak value, root mean square, rise time, fall time, positive pulse width, or negative pulse width of the waveform of Channel 1 or Channel 2.

First, the properties `MeasurementType`, `Source`, `Value`, and `Units` will be added to the `Measurement` group object.

1. Expand the `Measurement` group node to display the group object's properties and methods.
2. Select the `Properties` node to define the `Measurement` group object properties.
3. Enter the property name `MeasurementType` in the **Add property** text field and click the **Add** button.
4. Enter the property name `Source` in the **Add property** text field and click the **Add** button.
5. Enter the property name `Value` in the **Add property** text field and click the **Add** button.
6. Enter the property name `Units` in the **Add property** text field and click the **Add** button.
7. Expand the `Properties` node to display the group object's properties.

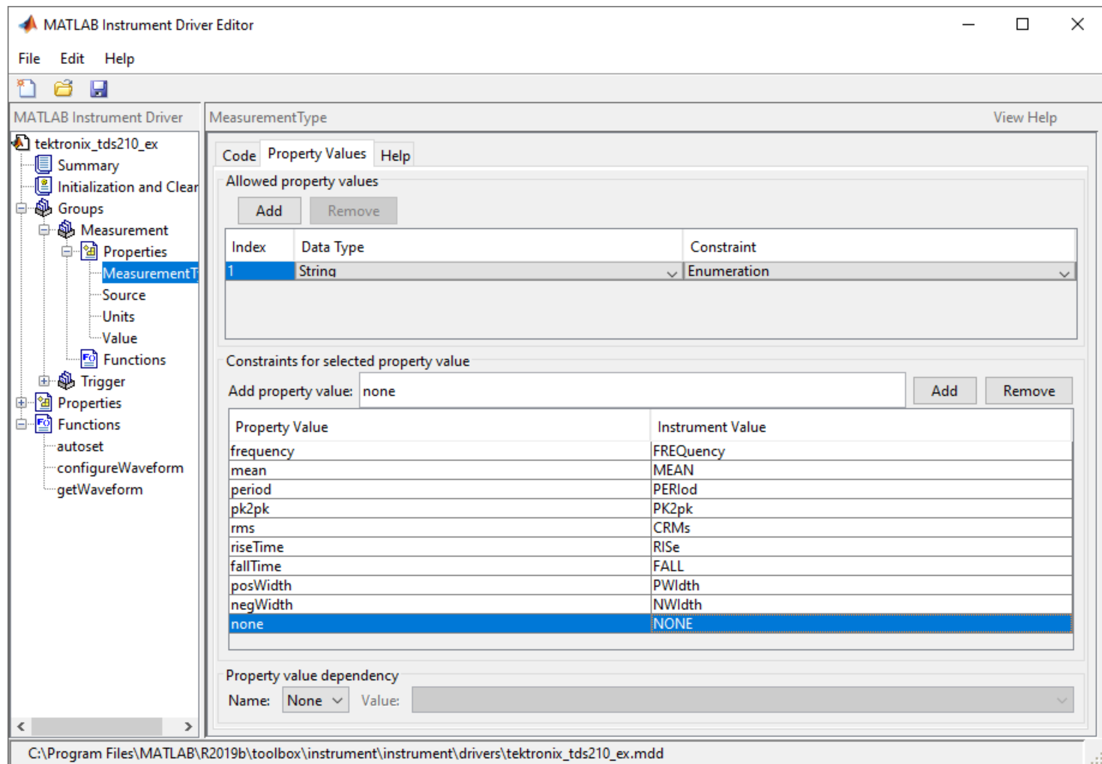


Next, define the behavior of the `MeasurementType` property:

1. Select the `MeasurementType` node in the tree.
2. Select the **Code** tab to define the `set` and `get` commands for the `MeasurementType` property.

   - Select `Instrument Commands` in the **Property style** field.
   - Enter `Measurement:<ID>:Type?` in the **Get command** text field.
   - Enter `Measurement:<ID>:Type` in the **Set command** text field.
3. Select the **Property Values** tab to define the allowed property values.

   - Select `String` in the **Data Type** field.
   - Select `Enumeration` in the **Constraint** field.
   - Enter `frequency` in the **Add property value** text field and click the **Add** button. Then enter `FREQuency` in the **Instrument Value** table field.
   - Add the enumeration: `mean`, `MEAN`.
   - Add the enumeration: `period`, `PERIod`.
   - Add the enumeration: `pk2pk`, `PK2pk`.
   - Add the enumeration: `rms`, `CRMs`.
   - Add the enumeration: `riseTime`, `RISe`.
   - Add the enumeration: `fallTime`, `FALL`.
   - Add the enumeration: `posWidth`, `PWIdth`.
   - Add the enumeration: `negWidth`, `NWIdth`.
   - Add the enumeration: `none`, `NONE`.



4 Select the **Help** tab to finish defining the property behavior.

- Enter none in the **Default value** text field.
- Select never in the **Read only** field.
- In the **Help text** field, enter `Specifies the measurement type`.

Next, define the behavior of the `Source` property.

1 Select the `Source` node in the tree.

2 Select the **Code** tab to define the `set` and `get` commands for the `Source` property.

- Select `Instrument Commands` in the **Property style** field.
- Enter `Measurement:<ID>:Source?` in the **Get command** field.
- Enter `Measurement:<ID>:Source` in the **Set command** field.

3 Select the **Property Values** tab to define the allowed property values.

- Select `String` in the **Data Type** field.
- Select `Enumeration` in the **Constraint** field.
- Enter `CH1` in the **Add property value** text field and click the **Add** button. Then enter `CH1` in the **Instrument Value** table field.
- Similarly add the enumeration: `CH2`, `CH2`.

4 Select the **Help** tab to finish defining the property behavior.

- Enter `CH1` in the **Default value** text field.
- Select `never` in the **Read only** field.
- In the **Help text** field, enter `Specifies the source of the measurement`.

Next, define the behavior of the Units property.

1 Select the Units node in the tree.
2 Select the **Code** tab to define the set and get commands for the Units property.
   - Select Instrument Commands in the **Property style** field.
   - Enter Measurement:<ID>:Units? in the **Get command** text field.
   - Since the Units property is read-only, leave the **Set command** text field empty.
3 Select the **Property Values** tab to define the allowed property values.
   - Select String in the **Data Type** field.
   - Select None in the **Constraint** field.
4 Select the **Help** tab to finish defining the property behavior.
   - Enter volts in the **Default value** text field.
   - Select always in the **Read only** field.
   - In the **Help text** field, enter Returns the measurement units.

Finally, define the behavior of the Value property.

1 Select the Value node in the tree.
2 Select the **Code** tab to define the set and get commands for the Value property.
   - Select Instrument Commands in the **Property style** field.
   - Enter Measurement:<ID>:Value? in the **Get command** text field.
   - Since the Value property is read-only, leave the **Set command** text field empty.
3 Select the **Property Values** tab to define the allowed property values.
   - Select Double in the **Data Type** field.
   - Select None in the **Constraint** field.
4 Select the **Help** tab to finish defining property behavior.
   - Enter 0 in **Default value** field.
   - Select always in the **Read only** field.
   - In the **Help text** field, enter Returns the measurement value.
5 Click the **Save** button.

Verifying the Properties of the Group Object in the MATLAB software

This procedure verifies the properties of the measurement group object. In this example, the driver name is tektronix_tds210_ex.mdd. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Measurement Computing Corporation GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, obj, using the icdevice function.

```
g = gpib('mcc',0,2);
obj = icdevice('tektronix_tds210_ex.mdd',g);
```

2 Extract the measurement group objects, m, from the device object.

```
m = obj.Measurement


  HwIndex:    HwName:    Type:                 Name:
```

```
   1           Meas1      scope-measurement    Measurement1
   2           Meas2      scope-measurement    Measurement2
   3           Meas3      scope-measurement    Measurement3
   4           Meas4      scope-measurement    Measurement4
```

3 View the current values for the properties of the first group object. Calling `get` on the object lists all its properties.

```
m(1)
```

```
   HwIndex:    HwName:    Type:                Name:
   1           Meas1      scope-measurement    Measurement1
```

4 Calling `get` on a specific property lists its current value.

```
m(1).MeasurementType
```

```
ans =

    'none'
```

```
m(1).Source
```

```
ans =

    'CH1'
```

```
m(1).Units
```

```
ans =

    'volts'
```

```
m(1).Value
```

```
ans =

     0
```

5 View the acceptable values for the properties of the group object. Calling `set` on the object lists all its settable properties.

```
set(m(1))
```

```
    Name:

    SCOPE-MEASUREMENT specific properties:
    MeasurementType: [ frequency | mean | period | pk2pk | rms | riseTime | fallTime | posWic
    Source: [ {CH1} | CH2 ]
```

```
set(m(1),'MeasurementType')
```

```
[ frequency | mean | period | pk2pk | rms | riseTime | fallTime | posWidth | negWidth | {none
```

```
set(m(1),'Source')
```

```
[ {CH1} | CH2 ]
```

6 Try setting the property to valid and invalid values.

```
m(1).Source = 'CH2'

   HwIndex:    HwName:    Type:                Name:
   1           Meas1      scope-measurement    Measurement1
   2           Meas2      scope-measurement    Measurement2
   3           Meas3      scope-measurement    Measurement3
   4           Meas4      scope-measurement    Measurement4
```

```
m(1).Source
```

```
ans =

    'CH2'
```

```
m(1).Source = 'CH5'
```

```
There is no enumerated value named 'CH5'.
```

7 View the help you wrote.

```
instrhelp(m(1),'Value')
```

```
   VALUE  (double)  (read only)

   Returns the measurement value.
```

8 List the group object characteristics that you defined in the **Property Values** and **Help** tabs.

```
propinfo(m(1),'Units')
```

```
ans =

  struct with fields:

               Type: 'string'
         Constraint: 'none'
    ConstraintValue: ''
       DefaultValue: 'volts'
           ReadOnly: 'always'
  InterfaceSpecific: 1
```

9 Connect to your instrument to verify the `set` and `get` code.

```
connect(obj)
```

Note When you issue the `get` function on the `MeasurementType` property for the first measurement object in the group, the `textronix_tds210_ex.mdd` driver actually sends the `Measurement:Meas1:Type?` command to the instrument.

```
m(1).MeasurementType
```

```
ans =

    'freequency'
```

Note When you issue the `set` function on the `Source` property for the second measurement object in the group, the `textronix_tds210_ex.mdd` driver actually sends the `Measurement:Meas2:Source CH2` command to the instrument.

```
m(2).Source = 'CH2';
```

10 Disconnect from your instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

Using Existing Drivers

In this section...

"Modifying MATLAB Instrument Drivers" on page 19-47
"Importing VXIplug&play and IVI Drivers" on page 19-47

Modifying MATLAB Instrument Drivers

If a MATLAB instrument driver does not exist for your instrument, it may be that a MATLAB instrument driver for an instrument similar to yours does exist. Rather than creating a new MATLAB instrument driver, you may choose to edit an existing MATLAB instrument driver. An existing MATLAB instrument driver can be opened in the MATLAB instrument driver editor with the `midedit` function.

```
midedit('drivername')
```

Deleting an Existing Property, Function, or Group

1. Select the property, function, or group in the tree.
2. Select the **Edit** menu.
3. Select the **Delete** menu item.

Renaming an Existing Property, Function, or Group

1. Select the property, function, or group in the tree.
2. Select the **Edit** menu.
3. Select the **Rename** menu item.

Other Settings and Tasks

Refer to "Creating MATLAB Instrument Drivers" on page 19-4 for information on

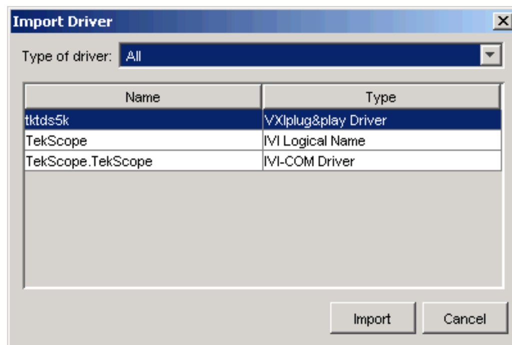
- Defining summary information
- Defining initialization and cleanup code
- Creating a new property
- Creating a new function
- Creating a new group

Importing VXIplug&play and IVI Drivers

The MATLAB Instrument Driver Editor can import a VXI*plug&play* or IVI driver. You can evaluate or set the driver's functions and properties, and the modified driver can be saved for further use:

1. Open the MATLAB Instrument Driver Editor with `midedit`.
2. Click the **File** menu, and select **Import**.

   The **Import Driver** dialog box appears, showing the installed VXI*plug&play* and IVI drivers.



3 Select the desired driver and click **Import**.

The MATLAB Instrument Driver Editor creates a MATLAB instrument driver based on the properties and/or functions in the original VXI*plug&play* or IVI driver. The editor displays the new driver's summary information, groups, properties, and functions.

With the MATLAB Instrument Driver Editor, you can

- Create, delete, modify, or rename properties, functions, or groups
- Add code around instrument commands for analysis
- Add create, connect, and disconnect code
- Save the driver as a separate MATLAB VXI*plug&play* instrument driver or MATLAB IVI instrument driver

Using the Instrument Driver Testing Tool

This chapter describes how to use the Instrument Driver Testing Tool to verify the functionality of your instrument drivers.

- "Instrument Driver Testing Tool Overview" on page 20-2
- "Setting Up Your Test" on page 20-4
- "Defining Test Steps" on page 20-9
- "Saving Your Test" on page 20-19
- "Testing and Results" on page 20-21

Instrument Driver Testing Tool Overview

In this section...

"Functionality" on page 20-2
"Drivers" on page 20-2
"Test Structure" on page 20-2
"Starting" on page 20-3
"Example" on page 20-3

Functionality

This section provides an overview of the MATLAB Instrument Driver Testing Tool and examples showing its capabilities and usage.

The MATLAB Instrument Driver Testing Tool provides a graphical environment for creating a test to verify the functionality of a MATLAB instrument driver.

The MATLAB Instrument Driver Testing Tool provides a way to do the following:

- Verify property behavior.
- Verify function behavior.
- Save the test as a test file, a MATLAB code, or driver function.
- Export the test results to MATLAB workspace, figure window, MAT-file, or the MATLAB Variables editor.
- Save test results as an HTML page.

Drivers

You can use the MATLAB Instrument Driver Testing Tool to test any MATLAB instrument driver, which include:

- MATLAB interface drivers
- MATLAB VXI*plug&play* drivers
- MATLAB IVI drivers

MATLAB VXI*plug&play* drivers and MATLAB IVI drivers can be created from VXI*plug&play* and IVI drivers, respectively, using the MATLAB Instrument Driver Editor or the `makemid` function.

Test Structure

The driver test structure is composed of setup information and test steps.

Setup

When setting up or initializing the test, you provide a test name and description, identify the driver to test, define the interface to the instrument, and set the test preferences. This information remains unchanged throughout the execution of the test, and applies to every step.

Test Steps

The executable portion of the test is divided into any number of test steps. A test step can perform one of four verifications:

- Set property — Verify that the set command or set code of a single device object or group object property in the driver does not error, and that the driver supports the defined range for the property value. You can use one value or all supported values for the property. You may also use invalid property values to check the driver's response.
- Get property — Verify the reading of a single device object or group object property from the driver.
- Properties sweep — Verify several properties in a single step.
- Function — Verify the execution of a driver function.

After configuring your test steps, you can execute the steps individually, or run a complete test that executes all the steps in the test.

Starting

You start the MATLAB Instrument Driver Testing Tool by typing the MATLAB command

```
midtest
```

This opens the tool without any test file loaded.

You may specify a test file (usually created in an earlier session of the tool) when you start the tool so that it opens up with a test file already loaded.

```
midtest('MyDriverTestfile')
```

Note MIDTEST and the Instrument Driver Testing Tool are unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

Example

For the examples in this chapter, you will create a test for the Tektronix TDS210 oscilloscope driver that you created in “MATLAB Instrument Driver Editor Overview” on page 19-2.

You will create each kind of step in your test: set property, get property, sweep properties, and function.

Setting Up Your Test

In this section...
"Test File" on page 20-4
"Providing a Name and Description" on page 20-4
"Specifying the Driver" on page 20-4
"Specifying an Interface" on page 20-4
"Setting Test Preferences" on page 20-4
"Setting Up a Driver Test" on page 20-5

Test File

You can specify a test file to load when you start `midtest`, open a test file after the MATLAB Instrument Driver Testing Tool is already up, or create a new test. You may find it convenient to keep the driver and test file together in the same directory. For easy use in the MATLAB Command Window, you can put that directory in the MATLAB path with the `addpath` command.

Note MIDTEST and the Instrument Driver Testing Tool are unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

Providing a Name and Description

The **Name** field allows a one-line text definition for your test. This name appears in the header of the test results in the Output Window.

The **Description** field allows a full definition of the text with as much descriptive text as you need.

Specifying the Driver

In the **Driver** text field, you specify the driver to be tested. This is any MATLAB instrument driver, usually with the `.mdd` extension. Enter the full path to the driver, or click **Browse** to navigate to the driver's directory.

Specifying an Interface

You specify the interface with the instrument for the testing of the driver. The instrument object type may be `GPIB`, `VISA`, `TCPIP`, `UDP`, or `serial port`. Depending on the type you choose, the **New Object Creation** dialog box prompts you for further configuration information.

The tool then creates a device object based on interface and driver.

Setting Test Preferences

The **Test Preferences** dialog box allows you to set certain behaviors of the tool when running a test.

Run Mode

This specifies whether the test runs all the steps or only one step in the test.

Fail Action

This specifies what happens if a step within the test fails. The test may stop after the failed step or continue, with or without resetting the instrument.

No-error String

This field specifies the expected string returned from the instrument *when there is no error*. If you indicate that a step passes when no error is returned from the instrument, the tool compares the string returned from the instrument via the `geterror` function, to the string given here in the Preferences dialog box. If the strings match, then the tool assumes there is no error from the instrument.

Number of Values to Test

A double-precision property can be tested using all supported values. You can request this when testing it as a single step, or the tool does it automatically when the property is tested as part of a property sweep step. This field specifies how many values are tested for such a property.

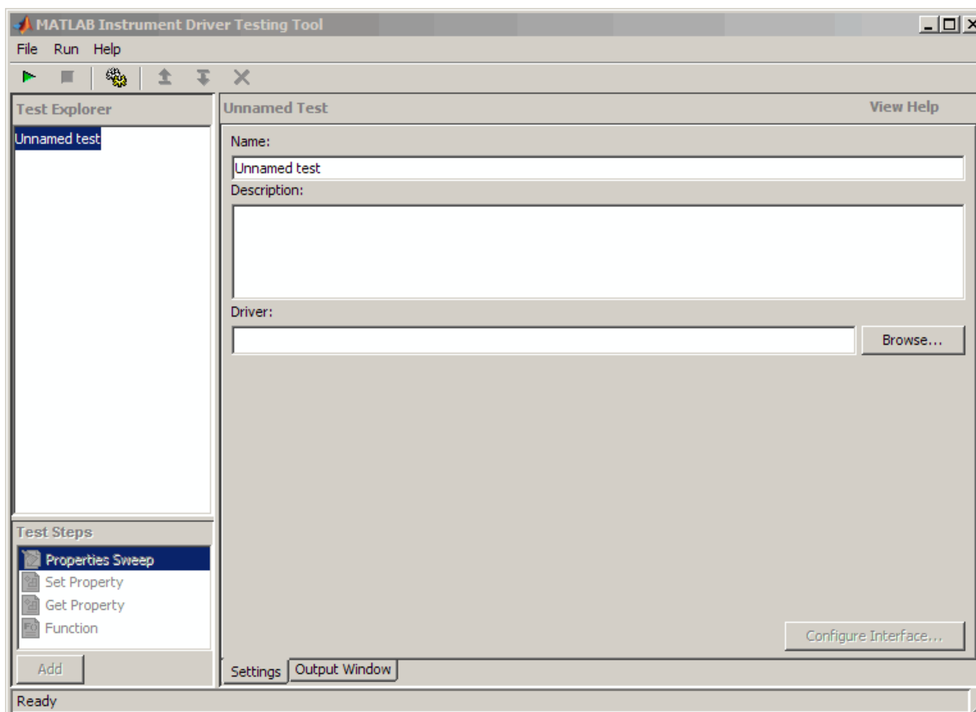
The number of values includes the defined minimum and maximum for the property, and integer values equally spaced between these limits.

If your property requires noninteger values for testing, then create a separate test step for that property instead of including it in a sweep.

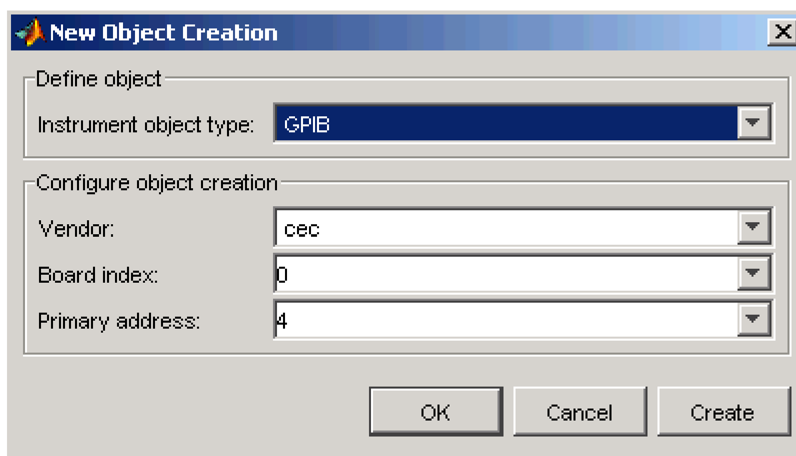
Setting Up a Driver Test

This example identifies the driver to be tested, and defines global setup information for the test. You will be testing the driver created in the examples of “MATLAB Instrument Driver Editor Overview” on page 19-2.

1 Open the MATLAB Instrument Driver Testing Tool from the command line with the command `midtest`.



2. In the **Name** text field, enter `TDS 210 Driver Sample Test`.
3. In the **Description** text field, enter `A test to check some of the properties and functions of the TDS 210 oscilloscope driver`.
4. In the **Driver** field, enter the name of the driver you created in “MATLAB Instrument Driver Editor Overview” on page 19-2. The text field will display the whole pathname, with the driver file `tektronix_tds210_ex.mdd`.
5. Click the **Create** button to create an instrument interface.



6. In the **New Object Creation** dialog box,

   a. Select your **Instrument object type**, **Vendor**, **Board index**, and **Primary address** of your instrument.

      The example illustrations in this chapter use a GPIB board with index 0 and the instrument at address 4. Your configuration may be different.

   b Click **OK**.

7. Click the **File** menu and select **Test Preferences**.
8. In the **Test Preferences** dialog box,

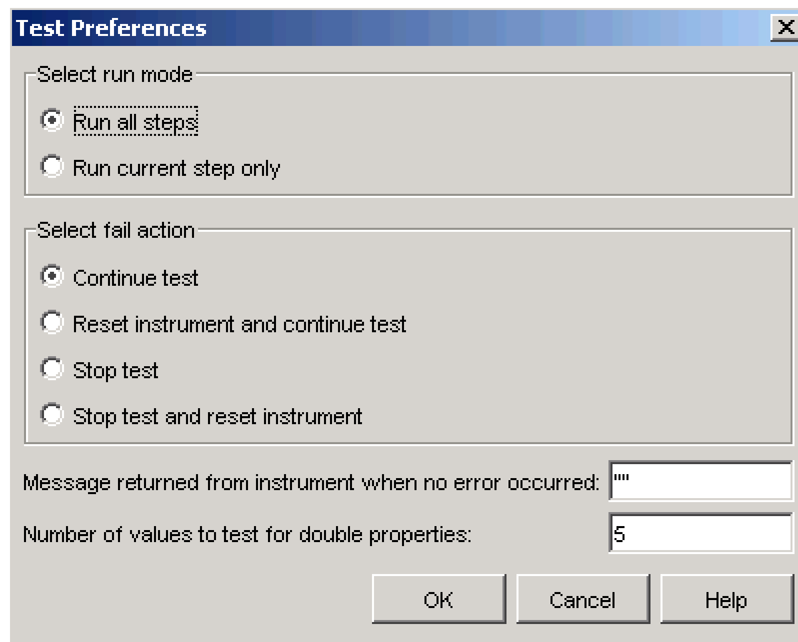
   a For **Select run mode**, click **Run all steps**.

   b For **Select fail action**, click **Continue test**.

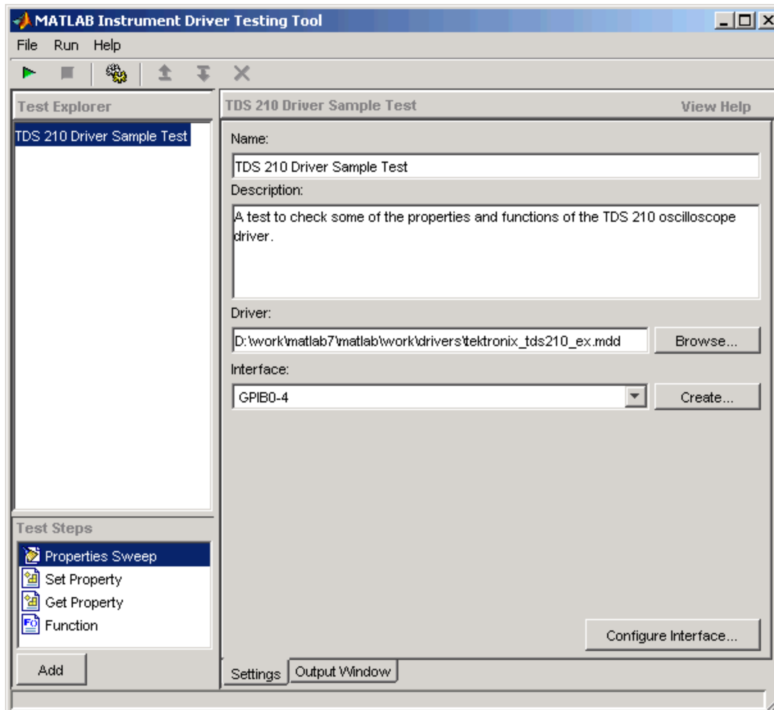
   c For **Message returned from instrument when no error occurred**, enter `""`. (This is an empty string in double quotes.)

   d For **Number of values to test for double properties**, enter `5`.

   e Click **OK**.



The MATLAB Instrument Driver Testing Tool now displays all your setup information.



9 Click **File** and select **Save**. Enter `tektronix_tds210_ex_test` as the filename for your test. The tool automatically adds the `.xml` file extension.

Defining Test Steps

In this section...

"Test Step: Set Property" on page 20-9
"Test Step: Get Property" on page 20-11
"Test Step: Properties Sweep" on page 20-13
"Test Step: Function" on page 20-15

Test Step: Set Property

You use a set property test step to verify a driver's set code or set command for a property. You provide a name for the step, select the driver property to test and the values to test it with, and define the conditions for the step's passing.

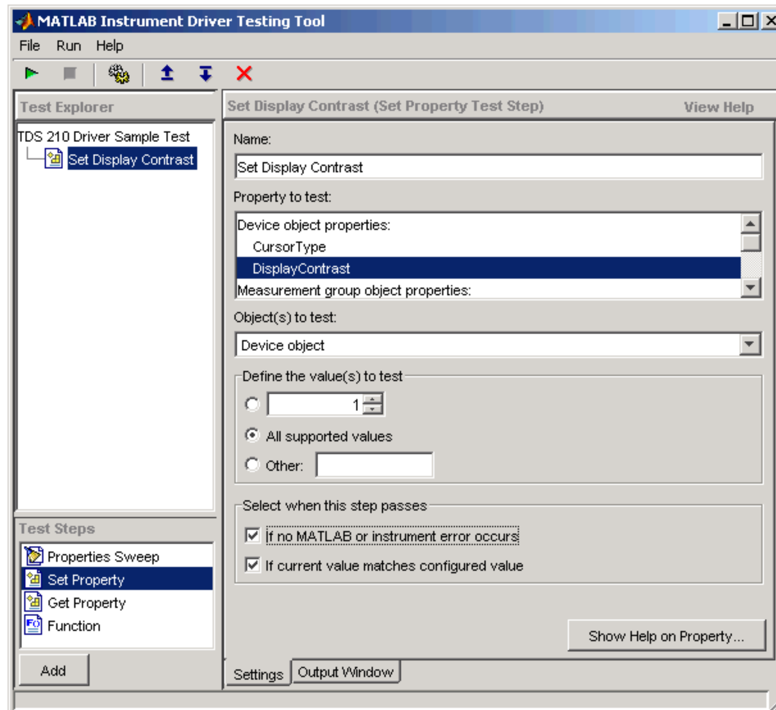
Settings	Dscription
Name	You provide a name for each test step. The name appears in the **Test Explorer** tree as well as in the results output.
Property to Test	A set property step can test only one property. You choose the property from the Property to Test list. Additional properties can be tested with additional steps, or with a sweep step.
Object(s) to Test	A property may be defined for the instrument or for a group object. If you are testing a group object property, you select which object you want tested in the Object(s) to Test list.
Define the Values to Test	If the property is has enumerated values, you can select one of the defined values, all of the supported values, or some other value. If the property's value is a double-precision number, you can select a value within its defined range, all supported values, or some other value. For a double, you set the number of values tested for all supported values in the Preferences dialog box (see "Number of Values to Test" on page 20-5).
Select When this Step Passes	The step passes when one or both of two conditions are met: • If no instrument or MATLAB error occurs as a result of attempting to set the property with its test value • If a query of the property after it is set returns a specified value If you select more than one of these conditions, then both conditions must be met for the step to pass. If no boxes are selected, the test will pass.

Creating a Test Step: Set Property

1. Click the `Set Property` option in the **Test Steps** list box.
2. Click the **Add** button.
3. In the **Name** field, enter `Set Display Contrast`.
4. In the **Property to test** list, select `DisplayContrast`.
5. For **Define the value(s) to test**, select `All supported values`.
6. For **Select when this step passes**,

   - Select **If no MATLAB software or instrument error occurs**.

- Select **If current value matches configured value**.

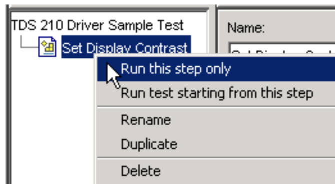


7 Click **File** and select **Save**.

Running a Test Step to Set a Property

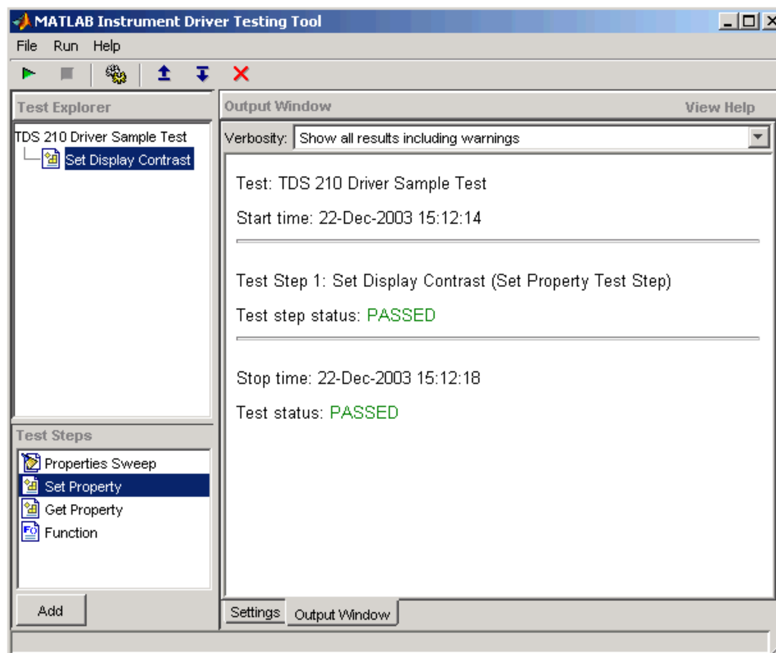
You can run an individual test step to verify its behavior:

1 Select `Set Display Contrast` in the **Test Explorer** tree.
2 With the cursor on the selected name, right-click to bring up the context menu.
3 In the context menu, select **Run this step only**.



You may want to repeat this step as you observe the oscilloscope display. The test sets the display contrast to five different values: lowest acceptable value (1%), highest acceptable value (100%), and three approximately equally spaced integer values between these limits.

The tool automatically displays the **Output Window** with the test results.



This test step passed because, for each of the five display contrast settings, the tool read back a value that was equal to the configured value.

Test Step: Get Property

You use a get property test step to verify a driver's ability to read a property. You provide a name for the step, select the driver property to test, and define the conditions for the step's passing.

Settings

The settings for the get property step are the same as for a "Test Step: Set Property" on page 20-9, except that instead of providing a value to write, you can provide an output argument variable.

Output Argument

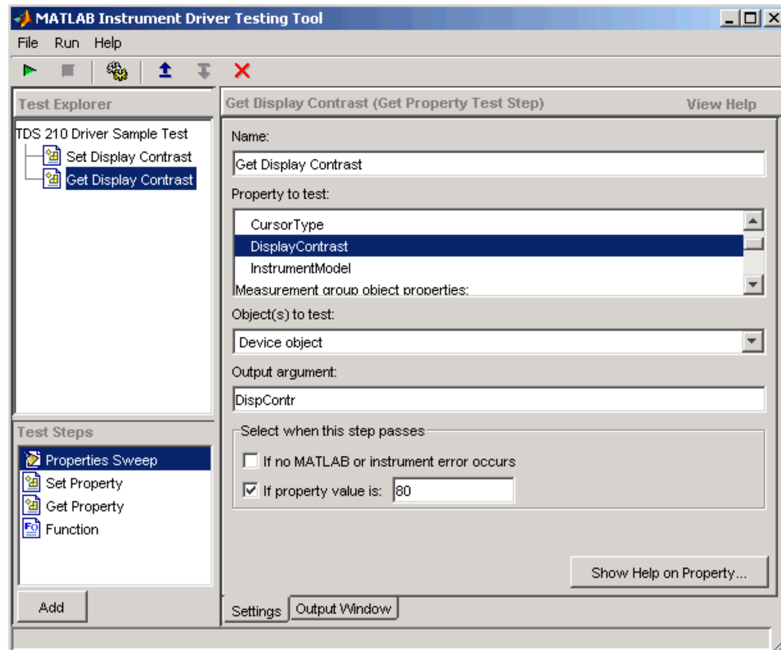
The test step assigns the optional output argument variable the value that results from reading the property. The variable is available for "Exporting Results" on page 20-22, after the test step has executed.

Creating a Test Step: Get Property

1. Click the `Get Property` option in the **Test Step** field.
2. Click the **Add** button.
3. In the **Name** field, enter `Getting Display Contrast`.
4. In the **Property to test** list, select `DisplayContrast`.
5. In the **Output argument** field, enter `DispContr`.
6. For **Select when this step passes**,

   - Unselect the box for **If no MATLAB software or instrument error occurs**.
   - Select **If property value is**, and enter a value of `80`.

This value is chosen to generate a failure. If this step follows the previous step in the example, the display contrast is still set at `100`. If this step is run by itself, the display contrast is set to `50` by the `*RST` command that is executed as part of your connect code for the driver.



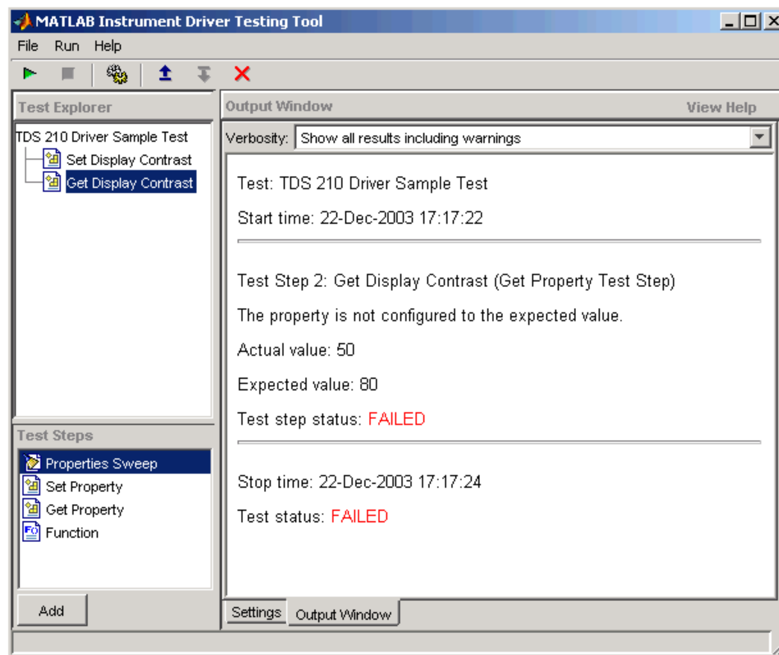
7 Click **File** and select **Save**.

Running a Test Step to Get a Property

You run the individual test step to verify its behavior.

1 Select `Get Display Contrast` in the **Test Explorer** tree.
2 With the cursor on the selected name, click the right mouse button to bring up the context menu.
3 In the context menu, select **Run this step only**.

Note that the test fails, reading a value of `50` while expecting a value of `80`.



Test Step: Properties Sweep

A properties sweep step allows you to test several properties in a single step. All selected properties are tested for all supported values. (In the case of properties with double-precision values, you determine the “Number of Values to Test” on page 20-5, in the **Test Preferences** dialog box.)

Settings

The fields for name and passing conditions are the same as other types of test steps. The sweep step also requires that you select which properties and groups to test.

Select the Properties to Test

You may select any or all of the properties for testing in a sweep step. You may find it convenient to create several sweep steps for testing related groups properties together.

Select the Group Object to Use on Sweep

For those properties defined for group objects, you can select a particular group object to test, or all the group objects. You can also define different sweep steps for different group objects.

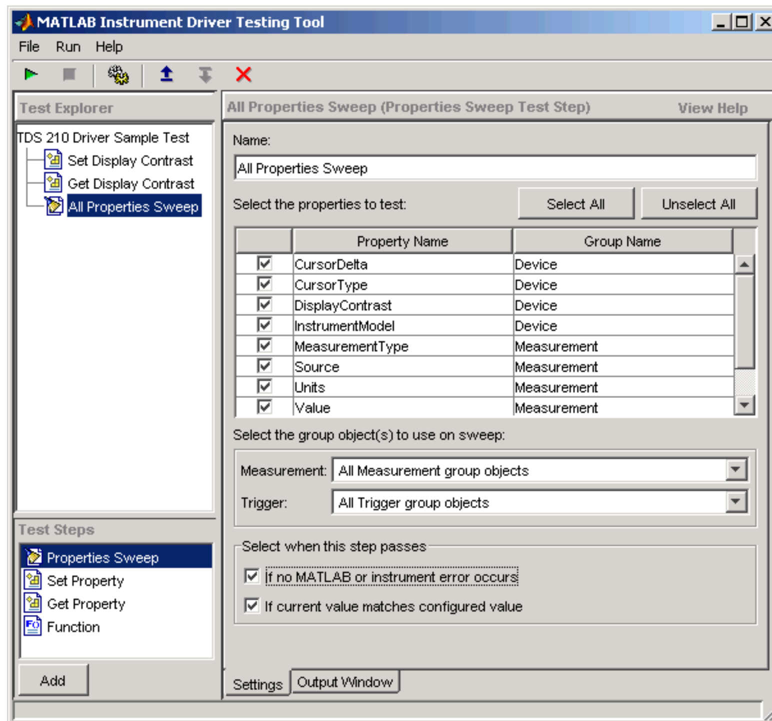
Creating a Sweep Step to Test All Properties

1. Click the `Properties Sweep` option in the **Test Step** field.
2. Click the **Add** button.
3. In the **Name** field, enter `All Properties Sweep`.
4. For **Select the properties to test**, click **Select All**.
5. In the Select the group object(s) field,

   - For the **Measurement** group, select `All Measurement group objects`.
   - For the **Trigger** group, select `All Trigger group objects`.
6. For **Select when this step passes**,

- Select **If no MATLAB software or instrument error occurs**, and
- Select **If current value matches configured value**

7 Click **File** and select **Save**.

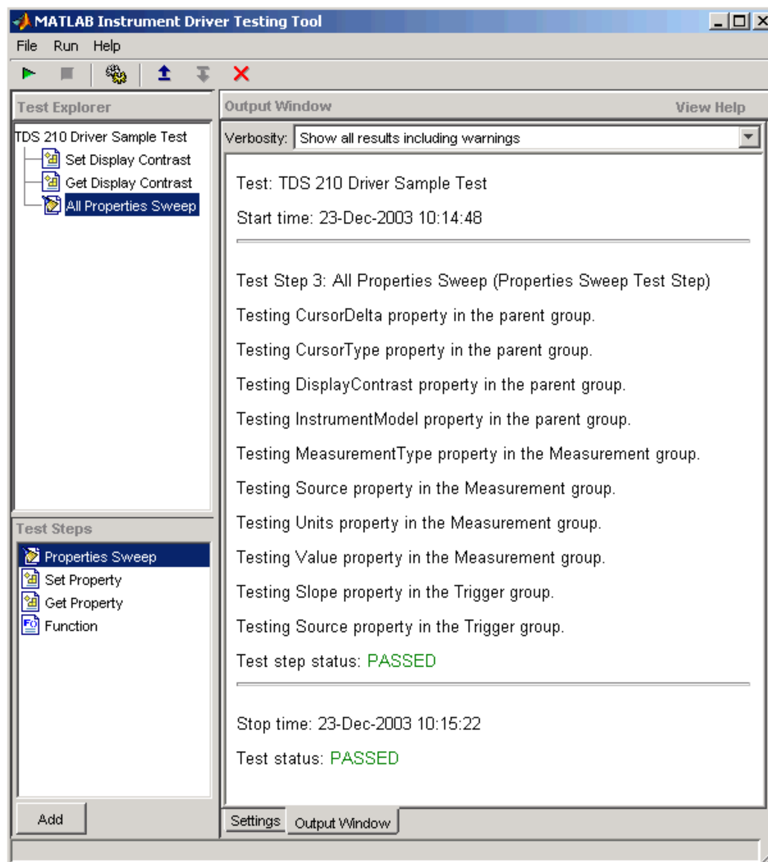


Running a Sweep Step to Test All Properties

You run the sweep test step to verify its behavior.

1 Select `All Properties Sweep` in the **Test Explorer** tree.
2 With the cursor on the selected name, click the right mouse button to bring up the context menu.
3 In the context menu, select **Run this step only**.

The **Output Window** is updated as each property in the sweep is tested. Note that the entire sweep is only one step in the overall test.



Test Step: Function

A function test step sends a function call to the instrument. You select the function called, the input data and output arguments (if required), and the conditions for passing.

Settings

Name

You provide a name for each test step. The name appears in the **Test Explorer** tree as well as in the results output.

Function to test

A function step can test only one function. You choose the function from the **Function to test** list. Additional functions can be tested with additional steps.

Function definition

The tool displays below the selected function what the call command for the function looks like. This helps you when deciding what input and output arguments to supply.

Input argument(s) and Output argument(s)

You provide input arguments as a comma-separated list of data, strings, character vectors, or whatever the function is expecting.

You provide output argument variable for any data returned from the function. The output arguments can be used to determine if the test step passes, or for “Exporting Results” on page 20-22 after the test step has executed.

Select when this step passes

The step passes when any of three conditions is met:

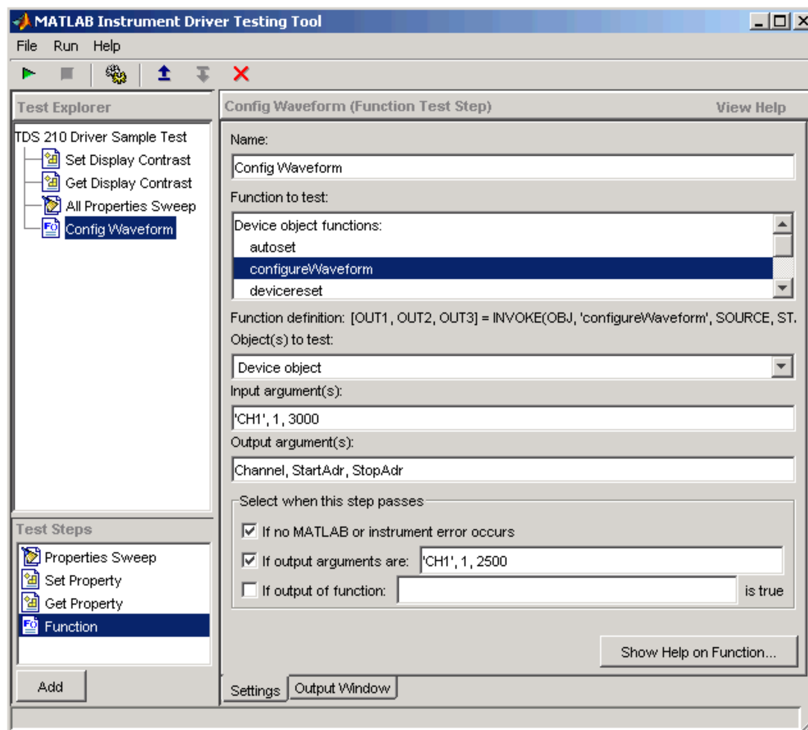
- If no instrument or MATLAB software error occurs as a result of attempting to execute the function
- If the returned output arguments match expected values
- If the output of a specified function is true

If you select more than one of these conditions, then all selected conditions must be met for the step to pass. If no boxes are selected, the test will pass.

Creating a Test Step: Function

1. Click the `Function` option in the **Test Step** field.
2. Click the **Add** button.
3. In the **Name** field, enter `Config Waveform`.
4. In the **Function to test** list, select `configureWaveform`.
5. In the **Input argument(s)** field, type `'CH1', 1, 3000`.
6. In the **Output argument(s)** field, type `Channel, StartAdr, StopAdr`.
7. For **Select when this step passes**,
   - Select **If no MATLAB software or instrument error occurs**.
   - Select **If output arguments are**, and enter in its field `'CH1', 1, 2500`.
   - Unselect **If output of function ... is true**.
8. Click **File** and select **Save**.

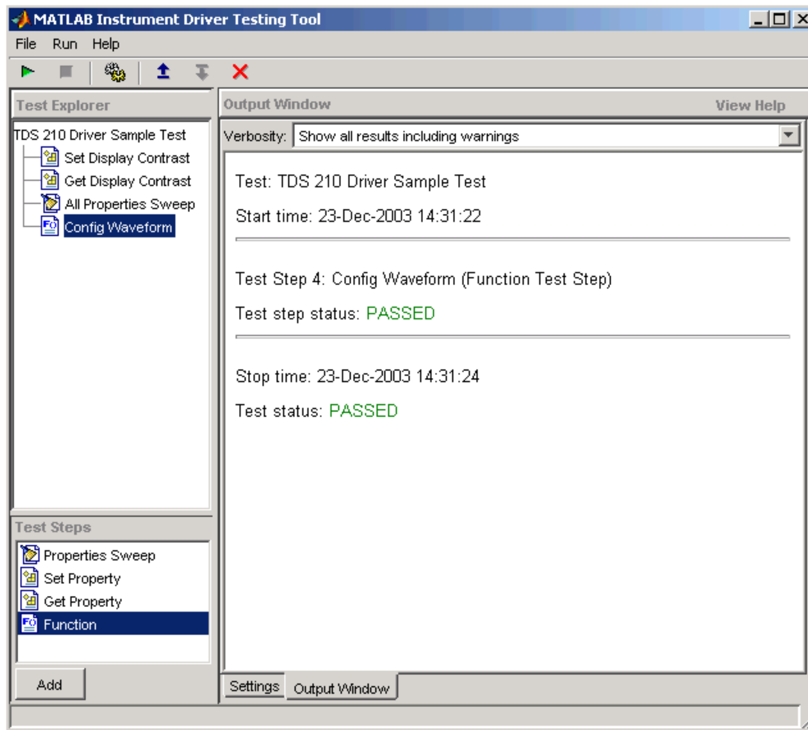
Note that you set the input argument for the stop address to `3000`, but you set the expected value for its output argument, `StopAdr`, to `2500`. This is because the maximum address of the oscilloscope is `2500`. If you attempt to exceed that value, the oscilloscope address is set to the maximum.



Running a Test Step to Test a Function

You can run an individual test step to verify its behavior

1. Select `Config Waveform` in the **Test Explorer** tree.
2. With the cursor on the selected name, click the right mouse button to bring up the context menu.
3. In the context menu, select **Run this step only**.


MATLAB Instrument Driver Testing Tool
File
Run
Help
Test Explorer
TDS 210 Driver Sample Test
Set Display Contrast
Get Display Contrast
All Properties Sweep
Config Waveform
Test Steps
Properties Sweep
Set Property
Get Property
Function
Add
Output Window
View Help
Verbosity:
Show all results including warnings
Test: TDS 210 Driver Sample Test
Start time: 23-Dec-2003 14:31:22
Test Step 4: Config Waveform (Function Test Step)
Test step status: PASSED
Stop time: 23-Dec-2003 14:31:24
Test status: PASSED
Settings
Output Window

Saving Your Test

In this section...
"Saving the Test as MATLAB Code" on page 20-19
"Saving the Test as a Driver Function" on page 20-19

Saving the Test as MATLAB Code

In the preceding examples of this chapter, you have been saving the test file after creating each step. The test file is saved in XML format. Here are some other save options.

You save the test file as MATLAB code by clicking the **File** menu and selecting **Save Test as M-Code**.

You can execute the test by calling this file from the MATLAB Command Window.

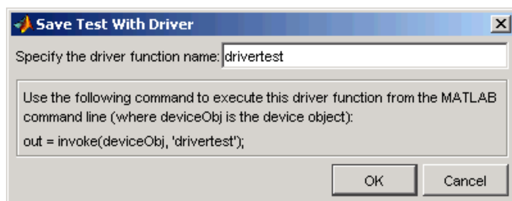
For example, you can save the test file you created in this chapter as `tektronix_tds210_ex_test.m`. Then you execute the test from the MATLAB Command Window by typing

```
tektronix_tds210_ex_test
```

The test results are displayed in the MATLAB Command Window.

Saving the Test as a Driver Function

You save your test as a driver function by clicking the **File** menu and selecting **Save Test as Driver Function**.



When you enter a name for the driver test function, the `invoke` command at the bottom of the dialog box reflects that name. You use that `invoke` command to execute the driver function from the MATLAB Command Window or in a file.

Creating a Driver Test Function

1. Click the **File** menu and select **Save Test as Driver Function**.
2. Enter `drivertest` in the **Specify the driver function name** field.
3. Click **OK**.

A function called `drivertest` is created and saved as part of the instrument driver file. You can open the driver file in the MATLAB Instrument Driver Editor tool (`midedit`) to verify that the `drivertest` function is included.

Calling a Driver Test Function from the MATLAB Command Window

Now that the test function is included in the driver, you access it with the `invoke` command from MATLAB.

In the MATLAB Command Window,

1. Create an interface object.

```
g = gpib('cec',0,4)
```

2. Create a device object, specifying the driver with the `drivertest` function saved in it.

```
obj = icdevice('tektronix_tds210_ex.mdd',g)
```

3. Connect to the device.

```
connect(obj)
```

4. Execute the driver test.

```
out = invoke(obj, 'drivertest')
```

5. When the test is complete, disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete ([g obj])
```

Testing and Results

In this section...
"Running All Steps" on page 20-21
"Partial Testing" on page 20-22
"Exporting Results" on page 20-22
"Saving Results" on page 20-23

Running All Steps

So far in this chapter, you have only run individual test steps after each was created.

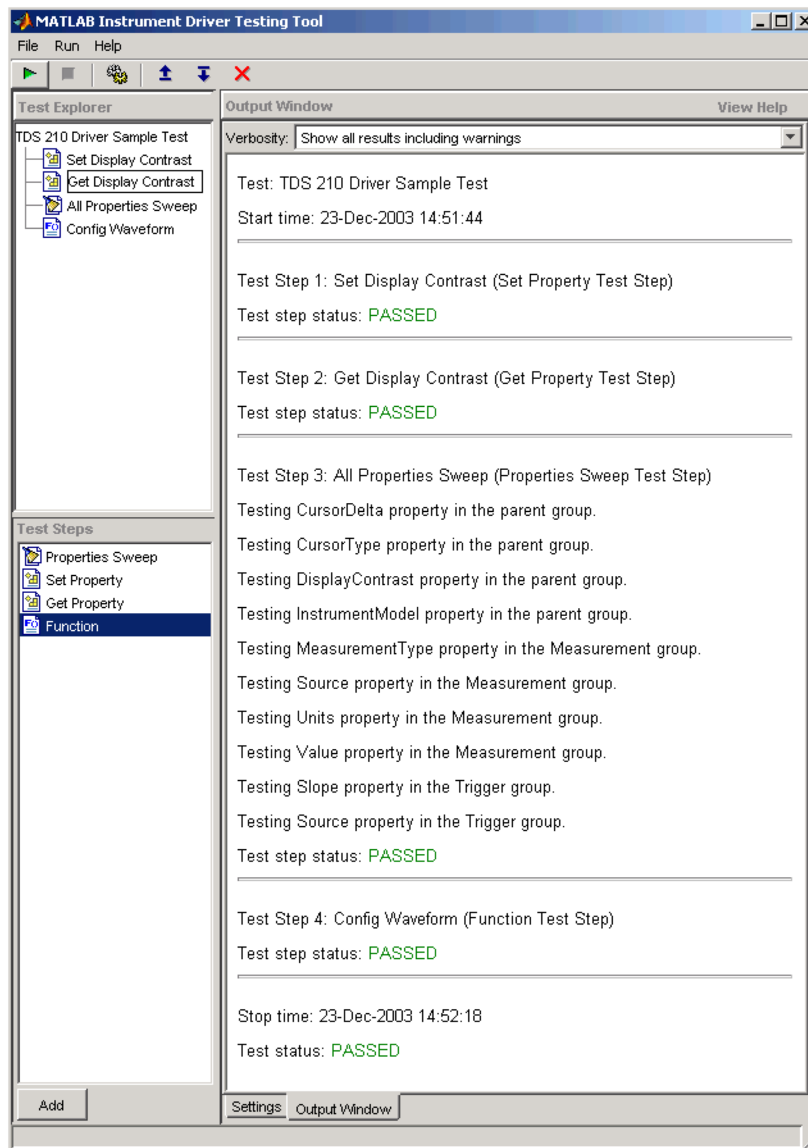
When you run the entire test, all the test steps run in the order listed in the **Test Explorer** tree. Using the mouse, you may drag the nodes of the tree to alter their sequence.

The **Output Window** displays the results of each step, along with a final result of the complete test.

Running a Complete Test

1. Select `Get Display Contrast` in the **Test Explorer** tree.
2. In the **Select when this step passes** field, change the **If property value is** entry from `80` to `100`.

   Earlier you entered a value of `80` to illustrate what a failure looks like. The display contrast is left at `100` from the `Set Display Contrast` test step, so that is what you will test for in the next step.
3. Click **File** and select **Save**.
4. Click the green arrow button to start a test run.



Partial Testing

Using the context menu in the **Test Explorer** tree, you can run a partial test of either an individual test step, or from the chosen test step through the end of the test.

Exporting Results

You can export the test results to many locations:

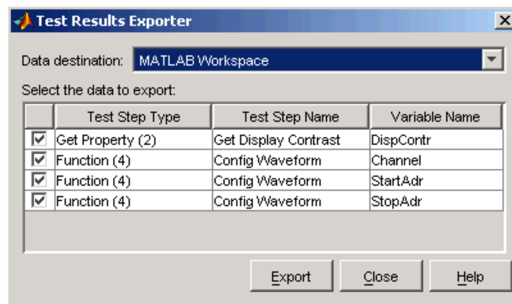
- MATLAB workspace
- MATLAB figure window
- MAT-file
- MATLAB Variables editor

The results you can export are those assigned to output variables in the settings for a test step.

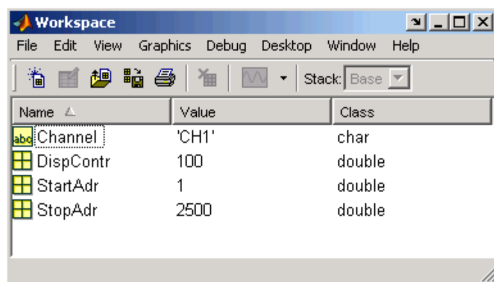
Exporting Test Results to the MATLAB Workspace

1. Click the **File** menu and select **Export Test Results**.
2. In the **Test Results Exporter** dialog box, select `MATLAB Workspace` as the **Data destination**.

   By default, all the variables are selected. You may unselect any.
3. Click the **Export** button.



The variables are now available in the MATLAB workspace, with values that were established by the test run.



Saving Results

You save your test results in an HTML file by clicking the **File** menu and selecting **Save Test Results**. The format of the results in this file reflects their appearance in the tester tool's **Output Window**.

Instrument Control Toolbox Troubleshooting

- “How to Use This Troubleshooting Guide” on page 21-3
- “Is My Hardware Supported?” on page 21-4
- “Troubleshooting SPI Interface” on page 21-6
- “Troubleshooting I2C Interface” on page 21-10
- “Troubleshooting MODBUS Interface” on page 21-14
- “Troubleshooting Serial Port Interface” on page 21-16
- “Troubleshooting GPIB Interface” on page 21-19
- “Troubleshooting TCP/IP Client Interface” on page 21-23
- “Troubleshooting TCP/IP Server Interface” on page 21-25
- “Troubleshooting UDP Interface” on page 21-27
- “Troubleshooting IVI and Quick-Control Interfaces” on page 21-29
- “Troubleshooting VISA Interface” on page 21-33
- “Hardware Support Packages” on page 21-36
- “Deploying Standalone Applications with Instrument Control Toolbox” on page 21-38
- “Contact MathWorks and Use the instrsupport Function” on page 21-41
- “Serial Warning - Unable to Read Any Data” on page 21-42
- “Serial Warning - Unable to Read All Data” on page 21-43
- “TCP/IP Warning - Unable to Read Any Data” on page 21-45
- “TCP/IP Warning - Unable to Read All Data” on page 21-46
- “UDP Warning - Unable to Read Any Data” on page 21-48
- “UDP Warning - Unable to Read All Data” on page 21-50
- “GPIB Warning - Unable to Read Any Data” on page 21-52
- “GPIB Warning - Unable to Read All Data” on page 21-53
- “TCP/IP Socket Using VISA Warning - Unable to Read Any Data” on page 21-55
- “TCP/IP Socket Using VISA Warning - Unable to Read All Data” on page 21-56
- “Bluetooth Warning - Unable to Read Any Data” on page 21-58
- “Bluetooth Warning - Unable to Read All Data” on page 21-59
- “Serialport Warning - Unable to Read Any Data” on page 21-61
- “Serialport Warning - Unable to Read All Data” on page 21-62
- “Resolve TCP/IP Client Warning: Unable to Read Any Data” on page 21-63
- “Resolve TCP/IP Server Warning: Unable to Read Any Data” on page 21-64
- “Resolve TCP/IP Server Warning: Unable to Read All Data” on page 21-65
- “Resolve UDP Port Warning: Unable to Read Any Data” on page 21-66

- “Resolve UDP Port Warning: Unable to Read All Data” on page 21-67
- “Resolve VISA Warning: Unable to Read Any Data” on page 21-68
- “Resolve VISA Warning: Unable to Read All Data” on page 21-70
- “Resolve Serial Port Connection Errors” on page 21-71
- “Resolve TCP/IP Client Connection Errors” on page 21-73
- “Resolve TCP/IP Server Connection Errors” on page 21-75
- “Resolve UDP Port Connection Errors” on page 21-76
- “Resolve VISA Connection Errors” on page 21-77

How to Use This Troubleshooting Guide

If you have trouble connecting to or communicating with an instrument, try the suggestions in this guide.

The first thing to check is that your instrument is supported with the toolbox. See "Is My Hardware Supported?" on page 21-4 for information about supported interfaces and supported hardware.

For connection and communication issues using a specific interface, see the section about that interface. For example, if you are having trouble using an instrument over the Bluetooth interface, refer to the Bluetooth section. Each interface section covers platform support, interface requirements, and troubleshooting tips and procedures for that interface.

If you are using VISA with another interface, try reading the sections for both interfaces. For example, if you are using VISA with UDP, try reading the sections on both VISA ("IVI, VISA, and the Quick Control Interfaces" topic) and UDP.

If you are having trouble with deployment or use of the MATLAB Compiler™, see "Deploying Standalone Applications with Instrument Control Toolbox" on page 21-38.

If you need to contact MathWorks Technical Support, read "Contact MathWorks and Use the instrsupport Function" on page 21-41 first. The `instrsupport` function runs diagnostics and provides useful information that may help solve your problem.

Is My Hardware Supported?

In this section...
“Supported Interfaces” on page 21-4
“Supported Hardware” on page 21-5

Supported Interfaces

The Instrument Control Toolbox supports the use of instruments and communication via the following interfaces. The table lists the interface support by platform. Notes after the table contain more specific information.

Feature	64-bit MATLAB on Windows	64-bit MATLAB on macOS	64-bit MATLAB on Linux
Serial	supported	supported	supported
TCP/IP	supported	supported	supported
UDP	supported	supported	supported
VISA [3]	supported [1]	supported on two vendors [1, 3]	
GPIB [4]	supported [1]		supported [1]
I2C [5]	supported [1]	supported [1]	supported [1]
SPI [5]	supported [1]	supported [1]	supported [1]
MODBUS	supported	supported	supported
Quick-Control Oscilloscope, Quick-Control Function Generator, Quick-Control RF Signal Generator	supported [2]	supported [2]	supported [2]
MATLAB Instrument Drivers	supported	supported	supported
MATLAB Instrument Drivers made using IVI-C drivers and Instrument Wrappers for IVI-C drivers	supported [1]		

Table Notes

1. Dependent on support by third-party vendor driver for the hardware on this platform.

2. Dependent on third-party vendor support of platform when using an IVI-driver with Quick-Control Oscilloscope or Quick-Control Function Generator.

3. Requires Keysight (formerly Agilent), National Instruments, Rohde & Schwarz R&S VISA, or TAMS VISA compliant with VISA specification 5.0 or higher for any platform. Only National Instruments

VISA and Rohde & Schwarz R&S VISA are supported on macOS. The other vendors' VISA support does not include macOS.

4. Requires Keysight (formerly Agilent), ICS Electronics, Measurement Computing (MCC), ADLINK Technology, or National Instruments hardware and driver.

5. Requires Aardvark or National Instruments hardware and driver.

Supported Hardware

See Hardware Support from Instrument Control Toolbox for a complete list of supported hardware.

Troubleshooting SPI Interface

In this section...
“Supported Platforms” on page 21-6
“Adaptor Requirements” on page 21-6
“Configuration and Connection” on page 21-7

Serial Peripheral Interface (SPI) is a synchronous serial data link standard that operates in full duplex mode. It is commonly used in the test and measurement field. Common uses include communicating with micro controllers, EEPROMs, A2D devices, embedded controllers, etc.

Instrument Control Toolbox SPI support lets you open connections with individual chips and to read and write over the connections to individual chips using an Aardvark or NI-845x host adaptor. The primary uses for the `spi` interface involve the `write`, `read`, and `writeAndRead` functions for synchronously reading and writing binary data.

Supported Platforms

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface.

The SPI interface is supported on these platforms when used with the Aardvark host adaptor:

- Linux — Red Hat Enterprise Linux 4 and 5 with kernel 2.6, and possibly SUSE and Ubuntu distributions.
- Microsoft Windows 64-bit

Note For R2018b and R2018a, you cannot use the Aardvark adaptor for I2C or SPI interfaces on the macOS platform. You can still use it on Windows and Linux. For releases prior to R2018a, you can use it on all three platforms, including the Mac.

The SPI interface is supported on these platforms when used with the NI-845x host adaptor:

- Microsoft Windows 64-bit

Adaptor Requirements

You need either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface. The following sections describe requirements for each option.

Aardvark-specific Requirements

To use the SPI interface with the Aardvark adaptor, download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the Aardvark driver installed, see “Install the Total Phase Aardvark I2C/SPI Interface Support Package” on page 15-9 to install it.

Install the Aardvark Software API and Shared Library appropriate for your operating system.

The `aardvark.dll` file that comes with the Total Phase Aardvark adaptor board must be available in one of these locations for use on Windows platforms:

- Location where MATLAB was started from (bin folder)
- MATLAB current folder (PWD)
- Windows folder `C:\winnt` or `C:\windows`
- Folders listed in the path environment variable

For use on Linux platforms, the `aardvark.so` file that comes with the Total Phase Aardvark adaptor board must be in your MATLAB path.

NI-845x-specific Requirements

To use the SPI interface with the NI-845x adaptor, download the hardware support package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this support package.

If you do not have the NI-845x driver installed, see “Install the NI-845x I2C/SPI Interface Support Package” on page 15-8 to install it.

Configuration and Connection

1. Make sure that you have the correct instrument driver installed for your device. Refer to your device’s documentation and the vendor’s web site.
2. Make sure your device is supported in Instrument Control Toolbox. See “Is My Hardware Supported?” on page 21-4.
3. You must have a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the SPI interface. Install the appropriate support package if you have not already. See “Adaptor Requirements” on page 21-6.

   Make sure that your SPI adaptor board is plugged into the computer running MATLAB. You can verify that you have one of the adaptors installed by using the `instrhwinfo` function with the `spi` interface name.

   ```
   >> instrhwinfo('spi')

   ans =

     HardwareInfo with properties:

       SupportedVendors: {'aardvark'  'ni845x'}
       InstalledVendors: {'ni845x'}
   ```

   If you do not see either `aardvark` or `ni845x` listed, you need to install one of the support packages or install the driver directly from the vendor.
4. Make sure that Instrument Control Toolbox recognizes your device, by using the `instrhwinfo` function with the `spi` interface name, and your adaptor name, either `aardvark` or `ni845x`. For example:

```
>> instrhwinfo('spi' , 'Aardvark')

ans =

            VendorName: 'aardvark'
     VendorDescription: 'Total Phase I2C/SPI Driver'
     VendorLibraryName: 'aardvark.dll'
     InstalledBoardIds: {[0]}
    BoardSerialNumbers: {'2237722838'}
    ObjectConstructors: {'spi('aardvark', 0, 0)'}
```

You will need the information displayed to create the `spi` object. If your device is not displayed, check the previous steps.

5 Make sure you can create the `spi` object. You must provide three arguments to create the object. `BoardIndex` and `Port` are both usually `0`, and `Vendor` is either `'aardvark'` or `'ni845x'`. This example uses a SPI object called `S` that communicates to an EEPROM chip. Create the object using the `BoardIndex` and `Port` numbers, which are `0` in both cases.

```
% Vendor = aardvark
% BoardIndex = 0
% Port = 0

S = spi('aardvark', 0, 0);
```

6 If you do not get an error, the object was created successfully. To verify, you can look at the object properties, using the name you assigned to the object, `S` in this case.

```
>> disp(S)
SPI Object :

    Adapter Settings
        BoardIndex:             0
        BoardSerial:            2237722838
        VendorName:             aardvark

    Communication Settings
        BitRate:                1000000 Hz
        ChipSelect:             0
        ClockPhase:             FirstEdge
        ClockPolarity:          IdleLow
        Port:                   0

    Communication State
        ConnectionStatus:       Disconnected

    Read/Write State
        TransferStatus:         Idle
```

7. Make sure you can connect to the device, using the `connect` function with the object name.

   ```
   connect(S);
   ```

   If you do not get an error, the connection was made successfully. If you do get an error, follow the steps in the error message and/or check the previous steps listed here.

8. When you have connected, you can communicate with your device. See “Transmitting Data Over the SPI Interface” on page 10-7 for an example of reading and writing to a chip.

Troubleshooting I2C Interface

In this section...
"Supported Platforms" on page 21-10
"Adaptor Requirements" on page 21-10
"Configuration and Connection" on page 21-11

I2C, or Inter-Integrated Circuit, is a chip-to-chip interface supporting two-wire communication. Instrument Control Toolbox I2C support lets you open connections with individual chips and to read and write over the connections to individual chips.

The Instrument Control Toolbox I2C interface lets you do chip-to-chip communication using an Aardvark or NI-845x host adaptor. Some applications of this interface include communication with SPD EEPROM and NVRAM chips, communication with SMBus devices, controlling accelerometers, accessing low-speed DACs and ADCs, changing settings on color monitors using the display data channel, changing sound volume in intelligent speakers, reading hardware monitors and diagnostic sensors, visualizing data sent from an I2C sensor, and turning on or off the power supply of system components.

Supported Platforms

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the I2C interface.

The I2C interface is supported on these platforms when used with the Aardvark host adaptor:

- Linux – The software works with Red Hat Enterprise Linux 4 and 5 with kernel 2.6. It may also be successful with SuSE and Ubuntu distributions.
- Microsoft Windows 64-bit

Note For R2018b and R2018a, you cannot use the Aardvark adaptor for I2C or SPI interfaces on the macOS platform. You can still use it on Windows and Linux. For releases prior to R2018a, you can use it on all three platforms, including the Mac.

The I2C interface is supported on these platforms when used with the NI-845x host adaptor:

- Microsoft Windows 64-bit

For updates to the list of currently supported platforms for MATLAB, see System Requirements.

Adaptor Requirements

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the I2C interface. The following sections describe requirements for each option.

Aardvark-specific Requirements

To use the I2C interface with the Aardvark adaptor, download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the Aardvark driver installed, see "Install the Total Phase Aardvark I2C/SPI Interface Support Package" on page 15-9 to install it.

Install the Aardvark Software API and Shared Library appropriate for your operating system.

The `aardvark.dll` file that comes with the Total Phase Aardvark adaptor board must be available in one of these locations for use on Windows platforms:

- Location where MATLAB was started from (bin folder)
- MATLAB current folder (PWD)
- Windows folder `C:\winnt` or `C:\windows`
- Folders listed in the path environment variable

For use on Linux platforms, the `aardvark.so` file that comes with the Total Phase Aardvark adaptor board must be in your MATLAB path.

NI-845x-specific Requirements

To use the I2C interface with the NI-845x adaptor, download the hardware support package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this support package.

If you do not have the NI-845x driver installed, see "Install the NI-845x I2C/SPI Interface Support Package" on page 15-8 to install it.

Configuration and Connection

1. Make sure that you have the correct instrument driver installed for your device. Refer to your device's documentation and the vendor's web site.
2. Make sure your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 21-4.
3. You must have a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `i2c` interface. Install the appropriate support package if you have not already. See "Adaptor Requirements" on page 21-10.

   Make sure that your I2C adaptor board is plugged into the computer running MATLAB. You can verify that you have one of the adaptors installed by using the `instrhwinfo` function with the `i2c` interface name.

   ```
   >> instrhwinfo('i2c')

   ans =

        InstalledAdaptors: 'Aardvark'
           JarFileVersion: 'Version 3.0.0'
   ```

   If you do not see either `aardvark` or `ni845x` listed, you need to install one of the support packages or install the driver directly from the vendor.

4 Make sure that Instrument Control Toolbox recognizes your device, by using the `instrhwinfo` function with the `i2c` interface name, and your adaptor name, either `aardvark` or `ni845x`. For example:

```
instrhwinfo('i2c', 'Aardvark')

ans =

          AdaptorDllName: [1x127 char]
       AdaptorDllVersion: 'Version 3.0.0'
             AdaptorName: 'aardvark'
       InstalledBoardIds: 0
   ObjectConstructorName: 'i2c('aardvark', BoardIndex, RemoteAddress);'
           VendorDllName: 'aardvark.dll'
 VendorDriverDescription: 'Total Phase I2C Driver'
```

You will need the information displayed to create the `i2c` object. If your device is not displayed, check the previous steps.

5 Make sure you can create the `i2c` object. You must provide three arguments to create the object. `BoardIndex` is usually `0`, and `Vendor` is either `aardvark` or `ni845x`. The `RemoteAddress` is specific to your device. Read the documentation of the chip in order to know what the remote address is. For example, in this case we create an object to communicate with an eeprom chip at remote address `50h`:

```
% Vendor = aardvark
% BoardIndex = 0
% RemoteAddress = 50

eeprom = i2c('aardvark', 0, '50h');
```

Tip You can also see what the remote address of the chip is by scanning for instruments in the Test & Measurement Tool. In the tool, right-click the **I2C** node and select **Scan for I2C adaptors**. Any chips found by the scan are listed in the hardware tree. The listing includes the remote address of the chip.

---

6 If you do not get an error, the object was created successfully. To verify, you can look at the object properties, using the name you assigned to the object, `eeprom` in this case.

```
>> disp(eeprom)

   I2C Object : I2C-0-50h

   Communication Settings
      BoardIndex          0
      BoardSerial         2237482577
      BitRate:            100
      RemoteAddress:      50h
      Vendor:             aardvark


   Communication State
      Status:             open
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
      BytesAvailable:     0
      ValuesReceived:     16
      ValuesSent:         15
```

7 Make sure you can connect to the device, using the `fopen` function with the object name.

```
fopen(eeprom);
```

If you do not get an error, the connection was made successfully. If you do get an error, follow the steps in the error message and/or check the previous steps listed here.

8 When you have connected, you can communicate with your device. See “Transmitting Data Over the I2C Interface” on page 9-6 for an example of reading and writing to a chip.

Troubleshooting MODBUS Interface

In this section...
“Supported Platforms” on page 21-14
“Configuration and Connection” on page 21-14
“Other Troubleshooting Tips for MODBUS” on page 21-15

Instrument Control Toolbox supports the MODBUS interface over TCP/IP or Serial RTU. You can use it to communicate with MODBUS servers, such as controlling a PLC (Programmable Logic Controller), communicating with a temperature controller, controlling a stepper motor, sending data to a DSP, reading bulk memory from a PAC controller, or monitoring temperature and humidly on a MODBUS probe.

Using the MODBUS interface, you can do the following tasks:

- Read coils, inputs, input registers, and holding registers
- Write to coils and holding registers
- Perform a combination of one write operation and one read operation on groups of holding registers in a single MODBUS transaction
- Modify the contents of a holding register using a mask write operation

Supported Platforms

Instrument Control Toolbox supports the MODBUS interface over TCP/IP or Serial RTU. It is supported on the following platforms.

- Linux 64-bit
- macOS 64-bit
- Microsoft Windows 64-bit

Note The Instrument Control Toolbox MODBUS support works on the MATLAB command line only. It is not available in the Test & Measurement Tool.

Configuration and Connection

1. Make sure your device is supported in Instrument Control Toolbox. See “Is My Hardware Supported?” on page 21-4.
2. If you are connecting to a local or remote device over MODBUS, make sure that the device is powered on and available.
3. Instrument Control Toolbox can communicate over MODBUS using TCP/IP or Serial RTU. If you are connecting via TCP/IP, you need to know the IP address or host name of the MODBUS server. If you are connecting via Serial RTU, you need to specify the Serial port the MODBUS server is connected to.
4. You can use the `instrhwinfo` function with the `modbus` interface name to see what Serial ports are available to use.

   ```
   instrhwinfo('modbus')
   ```

```
ans = 

HardwareInfo with properties:

      SupportedProtocols: ["serialrtu"    "tcpip"]
               LocalHost: [1×6 string]
    AvailableSerialPorts: ["COM1"    "COM3"]
```

In this case, `COM1` and `COM3` are available.

5. Make sure you can create the `modbus` object. You must provide arguments to create the object, whether you use `serialrtu` or `tcpip`. For examples of creating the object and information about the required arguments, see “Create a MODBUS Connection” on page 11-3.

   When you create the `modbus` object, it connects you to the server or device. There is no separate connection function required.
6. When you have connected, you can communicate with your device. See “Read Temperature from a Remote Temperature Sensor” on page 11-13 for an example of communicating with a device. See “Other Troubleshooting Tips for MODBUS” on page 21-15 for tips about communication issues after initial connection.

Other Troubleshooting Tips for MODBUS

These tips may be relevant to your use of the MODBUS interface.

Address Range

When specifying read and write addresses, the addresses must be in the following range: 0–65535.

Underlying Interface

You may encounter connection problems that are due to the underlying TCP/IP or Serial Port connections, rather than being specific to the MODBUS interface. The troubleshooting sections on “Troubleshooting TCP/IP Client Interface” on page 21-23 and “Troubleshooting Serial Port Interface” on page 21-16 might provide tips that will help.

MODBUS Addresses

If you have trouble figuring out a MODBUS address, see the vendor documentation of the device. For example, you may need to map a PLC register to the MODBUS address for the register. The vendor documentation may help.

Some vendors include an extra digit in addresses that gets dropped. For example 43233 is really address 3233. Devices are usually represented by a four-digit address, and some vendors use a 5th digit to represent the type of target, for example, coils. So you may need to adjust an address to account for this if your device vendor does that.

The Instrument Control Toolbox MODBUS functions use 1-based addressing, like the PLC shows addresses, not 0-based addressing, like MODBUS uses. The toolbox subtracts 1 from any addresses that are passed in via the address parameters in the read and write functions.

Troubleshooting Serial Port Interface

Serial communication is a low-level protocol for communicating between two or more devices. Normally, one device is a computer, and the other device can be another computer, modem, printer, or scientific instrument such as an oscilloscope or a function generator.

The serial port sends and receives bytes of information in a serial fashion — 1 bit at a time. These bytes are transmitted using either a binary format or a text (ASCII) format.

For many serial port applications, you can communicate with your instrument without detailed knowledge of how the serial port works. Communication is established through a serial port object, which you create in the MATLAB workspace.

Issue

If you are having trouble connecting to or communicating with your serial port device, follow these troubleshooting steps.

Possible Solutions

Supported Platforms

The serial port interface is supported on these platforms:

- Linux 64-bit
- macOS 64-bit
- Microsoft Windows 64-bit

The serial port interface is supported on the same platforms as MATLAB. For updates to the list of currently supported platforms, see System Requirements for MATLAB.

Adaptor Requirements

Use RS-232 interface standard with the serial port communication. Over the years, several serial port interface standards for connecting computers to peripheral devices have been developed. These standards include RS-232, RS-422, and RS-485 — all of which are supported by the serial port object. Of these, the most widely used standard is RS-232, which stands for Recommended Standard number 232.

You need to connect the two devices with a serial cable. For more information, see “Connecting Two Devices with a Serial Cable” on page 6-3.

Serial ports consist of two signal types: *data signals* and *control signals*. To support these signal types, as well as the signal ground, the RS-232 standard defines a 25-pin connection. However, most PCs and UNIX platforms use a 9-pin connection. In fact, only three pins are required for serial port communications: one for receiving data, one for transmitting data, and one for the signal ground. For more information, see “Serial Port Signals and Pin Assignments” on page 6-4.

Configuration and Connection

1. Make sure that you have the correct instrument driver installed for your device. Refer to your device documentation and the vendor website.

2. Make sure that your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 21-4.
3. Make sure that Instrument Control Toolbox recognizes your serial ports, by using the `serialportlist` function. For example, if your computer has more than one serial port, your output would look like this:

```
serialportlist

ans = 

  1×3 string array

    "COM1"    "COM3"    "COM4"
```

Tip You can also use Windows device manager to see a list of available serial ports.

4. Make sure you can create your serial port object. You must provide two arguments to create the object. For example, create a serial object called `s` using port `COM1` and baud rate `9600`.

```
s = serialport("COM1",9600);
```

   If you do not get an error, the object was created successfully.
5. When you have connected, you can communicate with your device. If you have problems sending or receiving, you may need to configure communication settings such as `BaudRate`, `DataBits`, `Parity`, `StopBits`, or `Terminator`. Make sure you configure these communication parameters to match those of the connected device.

   See "Writing and Reading Text Data" on page 6-17 and "Writing and Reading Binary Data" on page 6-19 for communication examples.

Other Troubleshooting Tips for Serial Port

Verify Port

Verify that the serial (COM) port is listed in Windows Control Panel > Device Manager > Ports.

Sending and Receiving

If you have problems sending or receiving, you may need to configure communication settings such as `BaudRate`, `DataBits`, `Parity`, `StopBits`, or `Terminator`. Make sure you configure these communication parameters to match those of the connected device.

VISA

For serial communication, you can also use VISA with a VISA resource name, as defined in a VISA vendor utility, such as Keysight Connection Expert.

Third-party Software

For troubleshooting serial port communication, you can also use a third-party serial communication software, such as PuTTY or Tera Term, to isolate the issue.

Incorrect Data

When doing binary data communication with `read` and `write`, make sure the correct data type – for example `int16`, `uint16`, `double` – is being used with `read` and `write`. You should use the same data type as the instrument uses.

If reading and writing data types other than `uint8` or `int8`, make sure the `ByteOrder` is correct.

See Also

`serialport` | `serialportlist`

Related Examples

- "Resolve Serial Port Connection Errors" on page 21-71
- "Serialport Warning - Unable to Read Any Data" on page 21-61
- "Serialport Warning - Unable to Read All Data" on page 21-62

Troubleshooting GPIB Interface

GPIB is a standardized interface that allows you to connect and control multiple devices from various vendors. GPIB is also referred to by its original name HP-IB, or by its IEEE designation IEEE-488. GPIB support in MATLAB is provided through the VISA-GPIB interface.

Issue

If you are having trouble connecting to or communicating with your GPIB device, follow these troubleshooting steps.

Possible Solutions

Supported Platforms and Minimum Driver Requirements

The VISA-GPIB interface is supported only on Microsoft Windows 10 (64-bit) and is not available for macOS or Linux.

The following table shows the minimum GPIB and VISA driver versions you must have. You must have one of the following GPIB drivers and its corresponding VISA driver both installed.

<table>
<tr><th>Minimum GPIB driver</th><th>Minimum VISA driver</th></tr>
<tr><td>Keysight IO Libraries version 18.1.24715.0 (Keysight Connection Expert 2019)</td><td rowspan="3">Keysight IO Libraries version 18.1.24715.0 (Keysight Connection Expert 2019)</td></tr>
<tr><td>ICS 488.2v4 Adaptor version 4.0</td></tr>
<tr><td>ADLINK ADL-GPIB version 20.01.0</td></tr>
<tr><td>NI-488.2 Adaptor v2.8</td><td rowspan="2">National Instruments NI-VISA version 19.5</td></tr>
<tr><td>MCC GPIB 488.2 Library v2.3</td></tr>
</table>

None of the supported GPIB drivers (Keysight, ICS, ADLINK, NI, and MCC) support asynchronous read and write operations.

The Keysight GPIB driver also has the following limitations:

- The End Or Identify (EOI) line is not asserted when the End-Of-String (EOS) character is written to the hardware.
- All eight bits are used for the EOS comparison.
- A board index value of `0` is not supported.
- An error is not reported for an invalid primary address. Instead, the read and write operations time out.

Configuration and Connection

1. Make sure that your adaptor is powered on. Make sure your device is also powered on.
2. Make sure that you have the correct GPIB and VISA drivers installed for your device. Refer to your device documentation and the vendor website.
3. Make sure your device is supported in Instrument Control Toolbox. See “Is My Hardware Supported?” on page 21-4 and “Instrument Control Toolbox Supported Hardware”.

4 Make sure that Instrument Control Toolbox recognizes your device, by using the `visadevlist` function.

```
resourceList = visadevlist
```

```
resourceList = 

  6×6 table

                     ResourceName                            Alias                Vendor
          _________________________________________    __________________    __________________

    1     "USB0::0x0699::0x036A::CU010105::0::INSTR"    "NI_SCOPE_4CH"        "TEKTRONIX"
    2     "TCPIP0::169.254.2.20::inst0::INSTR"          "Keysight_33210A"     "Agilent Technolo
    3     "ASRL1::INSTR"                                "COM1"                ""
    4     "ASRL3::INSTR"                                "COM3"                ""
    5     "GPIB0::5::INSTR"                             "FGEN_2CH"            "Agilent Technolo
    6     "GPIB0::11::INSTR"                            "OSCOPE_2CH"          "TEKTRONIX"
```

If the instrument is not listed, it might not be configured properly in your vendor’s VISA configuration utility software.

5 Make sure you can create the VISA-GPIB object using `visadev`. Each VISA-GPIB object is associated with one controller and one instrument. `visadev` requires the resource name or alias as an input. For example, create a VISA-GPIB object connected to a National Instruments controller with board index 0 and a Tektronix TDS1002 digital oscilloscope with primary address 1 and secondary address 0.

```
visagpib = visadev("GPIB0::1::0::INSTR")
```

If you do not get an error, the object was created successfully. If the resource name or alias does not exist, you get an error. Check that the resource name or alias is correct in the vendor configuration utility software.

You can have only one `visadev` object for a given resource at a time.

6 When you have connected, you can communicate with your device. If you have problems sending or receiving, you might need to configure communication settings. Make sure you are using the correct instrument command. Look in the instrument’s documentation to see what commands it recognizes. Verify that communication works by testing the connection using the vendor's configuration utility.

VISA Driver Configuration

If you are still having connection or communication issues with your GPIB resource using VISA, you can troubleshoot using your VISA vendor's software and utilities, as described in the following table.

Vendor	Configuration Utility	Testing Connection	Debug Utility
Keysight VISA	Keysight Connection Expert (KCE)	Interactive IO button on KCE	IO Monitor button on KCE
National Instruments VISA	Measurement and Automation Explorer (MAX)	Tools > NI VISA > VISA Interactive Control	Tools > NI I/O Trace

1 If you have multiple VISA installations, make sure that you have a preferred VISA set and that it is enabled. Check if all the VISA interfaces are using the expected VISA.

2. Use your vendor's configuration utility to make sure that your device hardware is being detected. Check that the hardware and interface properties are assigned as expected. You can also check that your device responds to a `*IDN?` query.
3. If you use SCPI commands, check if your device responds to them as expected when issued from the configuration utility.
4. Use your vendor's debug utility to check the Instrument I/O traffic for errors other than timeout errors.

Communication

1. Make sure the correct data type is being used with `read`, `readbinblock`, `write`, and `writebinblock`. Use the same data type that your instrument is configured to return.
2. If reading and writing data types other than `uint8` or `int8`, make sure the `ByteOrder` property is correct. You can configure it to be `little-endian` or `big-endian`. For GPIB, `ByteOrder` refers to the order in which the bytes in a multi-byte data type values are transmitted on the communication bus. You can use the `swapbytes` function to troubleshoot issues with `ByteOrder`. Configure `ByteOrder` to the appropriate value for your instrument before performing a read or write operation. Refer to your instrument documentation for information about the order in which it stores bytes.
3. `EOIMode` is the default and should be left `on` most of the time. However, some instruments might require `EOIMode` to be turned `off`.

Adaptor Requirements

Bus and connector

You need a bus and connector to communicate with GPIB instruments. The GPIB bus is a cable with two 24-pin connectors that allow you to connect multiple devices to each other. For more information, see “Bus and Connector” on page 4-2.

GPIB devices

Each GPIB device must be some combination of a Talker, a Listener, or a Controller. A Controller is typically a board that you install in your computer. Talkers and Listeners are typically instruments such as oscilloscopes, function generators, multimeters, and so on. Most modern instruments are both Talkers and Listeners. Each Controller is identified by a unique board index number. Each Talker/Listener is identified by a unique primary address ranging from 0 to 30, and by an optional secondary address, which can be 0 or can range from 96 to 126. For more information, see “GPIB Devices” on page 4-3.

GPIB data

Two types of data can be transferred over GPIB, instrument data and interface messages:

- Instrument data — Instrument data consists of vendor-specific commands that configure your instrument, return measurement results, and so on. For a complete list of commands supported by your instrument, refer to its documentation.
- Interface messages — Interface messages are defined by the GPIB standard and consist of commands that clear the GPIB bus, address devices, return self-test results, and so on.

  Data transfer consists of one byte (8 bits) sent in parallel. The data transfer rate across the interface is limited to 1 megabyte per second. However, this data rate is usually not achieved in practice, and is limited by the slowest device on the bus.

GPIB lines

GPIB consists of 24 lines, which are shared by all instruments connected to the bus. 16 lines are used for signals, while eight lines are for ground. The signal lines are divided into these groups:

- Eight data lines
- Five interface management lines
- Three handshake lines

For information on the types of lines and GPIB pin and signal assignments, see "GPIB Lines" on page 4-3.

See Also

`visadevlist` | `visadev`

Related Examples

- "Troubleshooting VISA Interface" on page 21-33

Troubleshooting TCP/IP Client Interface

Transmission Control Protocol (TCP) is a transport protocol layered on top of the Internet Protocol (IP) and is one of the most highly used networking protocols. You can use network socket communication to connect to remote hosts to read and write data.

Issue

If you are having trouble connecting to or communicating with your remote host, try these troubleshooting tips.

Possible Solutions

Supported Platforms

TCP/IP is supported on these platforms:

- Linux
- macOS
- Windows 10

The TCP/IP client interface is supported on the same platforms as MATLAB. For updates to the list of currently supported platforms, see System Requirements.

Configuration and Connection

1. Make sure you can create your TCP/IP client object. You create a client object with the `tcpclient` function, which requires the name of the remote host as an input argument. You also need to specify the remote port value.

   Each client object is associated with one instrument. For example, to create a client object for a Sony/Tektronix AWG520 Arbitrary Waveform Generator, you use the remote host name of the instrument and the port number, which can be found in the instrument documentation.

   ```
   t = tcpclient("sonytekawg.yourdomain.com",4000);
   ```

2. After you connect to the device, you can communicate with it. If sending and receiving does not work, you can check the following:

   - Make sure the data is being sent in the format expected by the server.
   - If you connect to a web server, you might need to send HTTP `get` or `post` commands. You can also use the `urlread` or `webread` functions to communicate with web servers.
   - Many TCP/IP servers expect header information inside the TCP/IP packet.

   See “Write and Read Data over TCP/IP Interface” on page 7-7 for an example of communication over TCP/IP.

VISA-TCP/IP Communication

You can also use the `visadev` interface with a VISA-TCP/IP resource name instead of the `tcpclient` interface for TCP/IP communication with instruments.

Incorrect Data Type

Make sure the correct data type—for example `int16`, `uint16`, `double`—is being used with `read` and `write`. Use the same data type as the instrument. If reading and writing data types other than `uint8` or `int8`, make sure the `ByteOrder` is correct.

See Also
`tcpclient`

Related Examples
- "Resolve TCP/IP Client Connection Errors" on page 21-73
- "Resolve TCP/IP Client Warning: Unable to Read Any Data" on page 21-63

Troubleshooting TCP/IP Server Interface

Transmission Control Protocol (TCP) is a transport protocol layered on top of the Internet Protocol (IP) and is one of the most highly used networking protocols.

Issue

If you are having trouble connecting to or communicating with your server, try these troubleshooting tips.

Possible Solutions

Supported Platforms

TCP/IP is supported on these platforms:

- Linux
- macOS
- Windows 10

The TCP/IP server interface is supported on the same platforms as MATLAB. For updates to the list of currently supported platforms, see System Requirements.

Configuration and Connection

Make sure you can create a TCP/IP server object. You create it using the `tcpserver` function, which takes a port number and an optional IP address as inputs. If you do not specify an IP address, the server can accept a client connection at any valid IP address. The IP address that you specify for the server should match the IP address that the client connects to.

```
t = tcpserver("172.28.200.145",4000);
```

1. Check that the IP address you specify is available on your machine. To see valid IP addresses for your machine, run the following command in MATLAB on Windows.

   ```
   !ipconfig
   ```

2. If you specify a host name instead of host IP address, you can also verify that it is a valid host name by using `resolvehost`. If the output is empty, the specified host name is invalid.

   ```
   resolvehost("hostname","address")


   ans =

       '172.28.200.145'
   ```

3. Make sure that you do not specify a port that is already in use. In addition, you can only create one server object for a given address and port combination.
4. Make sure that your network adapter is enabled and connected.

Communication

1. Before reading from or writing to the server object, check the `Connected` property to make sure that a client is connected to it. If a client has successfully connected, the value of this property is `1` (`true`).

   ```
   t.Connected
   ```

   The `ClientAddress` and `ClientPort` properties also provide information about the client that is connected to the server object. The value of `ClientAddress` should match the value of `ServerAddress`.
2. Make sure the correct data type, such as `int16`, `uint16`, or `double` is being used with `read` and `write`. Use the same data type as the client. If reading and writing data types other than `uint8` or `int8`, make sure the `ByteOrder` is correct.

See Also
`tcpserver`

Related Examples
- "Resolve TCP/IP Server Connection Errors" on page 21-75
- "Resolve TCP/IP Server Warning: Unable to Read Any Data" on page 21-64
- "Resolve TCP/IP Server Warning: Unable to Read All Data" on page 21-65

Troubleshooting UDP Interface

In this section...
“Issue” on page 21-27
“Possible Solutions” on page 21-27

User Datagram Protocol (UDP or UDP/IP) is a transport protocol layered on top of the Internet Protocol (IP). UDP is a connectionless protocol. An application using UDP prepares a packet and sends it to the receiver's address without first checking to see if the receiver is ready to receive a packet. If the receiving end is not ready to receive a packet, the packet is lost.

Issue

If you are having trouble connecting to or communicating with your UDP socket, follow these troubleshooting tips.

Possible Solutions

Supported Platforms

UDP is supported on these platforms:

- Linux
- macOS
- Windows 10

The UDP interface is supported on the same platforms as MATLAB. For updates to the list of currently supported platforms, see System Requirements.

The `configureMulticast` function is only supported on Windows and not supported Linux or macOS.

Configuration and Connection

1 Make sure you can create your UDP object with the `udpport` function.

   Although UDP is a stateless connection, creating a UDP object with an invalid local host or local port generates an error. Specifying a local port that is in use elsewhere or a port with port sharing disabled also generates an error. You can configure property values during object creation, such as the `LocalPort` property if you will use the object to read data. For example, create a `udpport` object associated with local port `3533`.

```
u = udpport("LocalPort",3533)
```

```
u = 

  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 3533
```

```
    NumBytesAvailable: 0

  Show all properties, functions
```

2. If the computer or host does not exist, you will get a warning. You can try to ping the computer to see if it is responding.

   ```
   !ping 127.0.0.1
   ```

3. When you have connected, you can communicate with your device. If sending and receiving does not work, you can check the following:

   - Make sure the destination address and destination port parameters for sending data with `write` and `writeline` are valid or exist. Try `!ping [destinationAddress]`.
   - Make sure the destination port and `LocalPort` are correct. The destination port is the port on the other computer to which data is sent. `LocalPort` is the port on the local computer that the `udpport` object is bound to.
   - UDP is not a reliable protocol and packets can be dropped. You may need to try sending or receiving multiple times.

   See "Write and Read ASCII Data Over UDP" on page 8-7 and "Write and Read Binary Data Over UDP" on page 8-9 for examples of communication over UDP and information on using properties.

See Also

`udpport`

Related Examples

- "Resolve UDP Port Connection Errors" on page 21-76
- "Resolve UDP Port Warning: Unable to Read Any Data" on page 21-66
- "Resolve UDP Port Warning: Unable to Read All Data" on page 21-67

Troubleshooting IVI and Quick-Control Interfaces

The IVI standard defines an open driver architecture, a set of instrument classes, and shared software components. With IVI you can use instruments interchangeability into multiple systems using standardized code.

You can use the Quick-Control Oscilloscope for any oscilloscope that uses an underlying IVI-C driver. You can use the Quick-Control Function Generator for any function generator that uses an underlying IVI-C driver. You can use the Quick-Control RF Signal Generator for any RF signal generator that uses an underlying IVI-C driver.

Supported Platforms

IVI is supported on these platforms:

- Microsoft Windows 64-bit

The Quick-Control interfaces are supported on these platforms:

- Microsoft Windows 64-bit

Adaptor Requirements

IVI-C

Instrument Control Toolbox software supports IVI-C drivers, with class-compliant and instrument-specific functionality.

IVI class-compliant drivers support common functionality across a family of related instruments. Use class-compliant drivers to access the basic functionality of an instrument, and the ability to swap instruments without changing the code in your application. With an IVI instrument-specific driver or interface, you can access the unique functionality of the instrument. The instrument-specific driver generally does not accommodate instrument substitution.

For IVI-C drivers, you can use IVI-C class drivers and IVI-C specific drivers. Device objects you construct to call IVI-C class drivers offer interchangeability between similar instruments, and work with all instruments consistent with that class driver. Device objects you construct to call IVI-C specific drivers directly generally offer less interchangeability, but provide access to the unique methods and properties of a specific instrument.

Other things to note:

- IVI-COM is no longer supported, because of the removal of 32-bit MATLAB.
- Using an IVI driver with `icdevice` requires generating a MATLAB instrument driver (MDD) with `makemid` or using a prebuilt MDD driver.
- The IVI Foundation maintains a registry of drivers sortable by instrument model and driver type. See http://www.ivifoundation.org/registered_drivers/driver_registry.aspx.

Before you use IVI drivers in MATLAB, install:

- VISA
- IVI Shared components

- Required IVI drivers

IVI-C Wrappers

The IVI-C wrappers provide an interface to MATLAB for instruments running on IVI-C class-compliant drivers.

To use the wrapper you must have the following software installed.

- Windows 64-bit
- VISA shared components
- VISA
- National Instruments compliance package NICP 4.1 or higher
- Your instrument driver

Quick-Control Oscilloscope

You can use the Quick-Control Oscilloscope for any oscilloscope that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. You can also use it for Tektronix oscilloscopes.

To use the Quick-Control Oscilloscope for an IVI-C scope, you must have the following software installed. Most components are installed by the Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces. To install the support package, see “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11.

- Windows 64-bit platforms
- VISA shared components (installed by the support package)
- VISA (installed by the support package)

  Note, the examples use Keysight VISA, but you can use any version of VISA.
- National Instruments IVI compliance package NICP 4.1 or later (installed by the support package)
- Your instrument’s device-specific driver. If you do not already have it, go to your instrument vendor's website and download the IVI-C driver for your specific instrument.

By default, the driver used is Tektronix (`'tektronix'`). If your instrument is not supported by the default driver, specify a particular IVI-C Scope driver using the `driver` property on the `oscilloscope` object.

Note As of release R2015a, most of these components are installed for you when you install the National Instruments VISA and ICP Interfaces Support Package. See “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11.

Quick-Control Function Generator

You can use the Quick-Control Function Generator for any function generator that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver.

To use the Quick-Control Function Generator for an IVI-C fgen, ensure the following software is installed. Most components are installed by the Instrument Control Toolbox Support Package for

National Instruments VISA and ICP Interfaces. To install the support package, see “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11.

- Windows 64-bit platforms
- VISA shared components (installed by the support package)
- VISA (installed by the support package)

  Note, the examples use Keysight VISA, but you can use any vendor’s implementation of VISA.
- National Instruments IVI compliance package NICP 4.1 or later (installed by the support package)
- Your instrument’s device-specific driver. If you do not already have it, go to your instrument vendor's website and download the IVI-C driver for your specific instrument.

By default, the driver used is `'Agilent332x0_SCPI'`. If your instrument is not supported by the default driver, specify a particular IVI-C Function Generator driver using the `driver` property on the `fgen` object.

Note As of release R2015a, most of these components are installed for you when you install the National Instruments VISA and ICP Interfaces Support Package. See “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11.

Quick-Control RF Signal Generator

You can use the Quick-Control RF Signal Generator for any RF signal generator that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver.

To use the Quick-Control RF Signal Generator for an IVI-C RF signal generator, ensure the following software is installed. Most components are installed by the Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces, but you can also install them separately. To install the support package, see “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11.

- Windows 64-bit platforms
- VISA shared components (installed by the support package)
- VISA (installed by the support package)

  Note, the examples use Keysight VISA, but you can use any vendor’s implementation of VISA.
- National Instruments IVI compliance package NICP 4.1 or later (installed by the support package)
- The device-specific driver for your instrument. If you do not already have it, go to your instrument vendor's website and download the IVI-C driver for your specific instrument.

Note As of release R2015a, most of these components are installed for you when you install the National Instruments VISA and ICP Interfaces Support Package. See “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11.

Configuration and Connection

1. Make sure that you have the correct instrument driver installed for your device. Refer to your device documentation and the vendor website.

2. Make sure that your device is supported in Instrument Control Toolbox. See “Is My Hardware Supported?” on page 21-4.
3. Make sure that Instrument Control Toolbox recognizes your device, by using the `instrhwinfo` function with the `ivi` interface name to find information on installed IVI drivers and shared components. For example:

```
instrhwinfo ('ivi')

ans =
              LogicalNames: {'MainScope', 'FuncGen'}
                ProgramIDs: {'TekScope.TekScope','Agilent33250'}
                   Modules: {'ag3325b', 'hpe363xa'}
ConfigurationServerVersion: '1.6.0.10124'
  MasterConfigurationStore: 'C:\Program Files\IVI\Data\
                             IviConfigurationStore.xml'
               IVIRootPath: 'C:\Program Files\IVI\'
```

   Modules refer to IVI-C drivers.

   Logical names are associated with particular IVI drivers as defined in the IVI Configuration Store, but they do not necessarily imply that the drivers are currently installed. You can install drivers that do not have a `LogicalName` property set yet, or drivers whose `LogicalName` was removed.

   Alternatively, use the Test & Measurement Tool to view the installation of IVI drivers and the setup of the IVI configuration store. Expand the `Instrument Drivers` node and click **IVI**. Click the **Software Modules** tab. (For information on the other IVI driver tabs and settings in the Test & Measurement Tool, see “IVI Configuration Store” on page 14-12.)
4. You can create an `ivi` object to communicate with your instrument. For instructions on creating an IVI object, constructing an IVI configuration store, and configuring communication using an IVI-C class compliant interface, see “Reading Waveforms Using the IVI-C Class Compliant Interface” on page 14-18.
5. When you have connected, you can communicate with your device. If you have problems sending or receiving, you may need to configure communication settings. Make sure you are using the correct instrument command. Look in the instrument’s documentation to see what commands it recognizes.

Troubleshooting VISA Interface

Virtual Instrument Software Architecture (VISA) is an industry standard defined by the IVI foundation for communicating with instruments regardless of the interface.

For the full VISA specifications maintained by the IVI Foundation, see IVI Specifications.

Issue

If you are having trouble connecting to or communicating with your VISA resource, follow these troubleshooting tips.

Possible Solutions

Supported Platforms and Minimum Driver Requirements

VISA is supported on these platforms:

- macOS (NI-VISA and R&S VISA only)
- Windows 10

These are the minimum VISA driver versions you must have:

- Keysight IO Libraries version 18.1.24715.0 (Keysight Connection Expert 2019)
- National Instruments NI-VISA version 19.5
- Rohde & Schwarz R&S VISA version 5.12

Tektronix TekVISA is not supported for the `visadev` interface.

Configuration and Connection

1. Make sure your device is powered on and all cables are properly connected.
2. Make sure that you have the correct instrument driver installed for your device. Refer to your device documentation and the vendor website.

   Note If you are connecting to a GPIB device using an NI GPIB adaptor, you must download the NI-488.2 driver compatible with your VISA driver version from the NI website. The NI-488.2 driver is not available as an Instrument Control Toolbox support package.
3. Make sure that your device is supported in Instrument Control Toolbox. See “Is My Hardware Supported?” on page 21-4 and “Instrument Control Toolbox Supported Hardware”.
4. Make sure that Instrument Control Toolbox recognizes your device, by using the `visadevlist` function.

```
resourceList = visadevlist
```

```
resourceList = 

  6×6 table

                   ResourceName                  Alias           Vendor
```

```
     ____________________________________________    __________________    ____________________

1    "USB0::0x0699::0x036A::CU010105::0::INSTR"    "NI_SCOPE_4CH"      "TEKTRONIX"
2    "TCPIP0::169.254.2.20::inst0::INSTR"          "Keysight_33210A"   "Agilent Technolo
3    "ASRL1::INSTR"                                "COM1"              ""
4    "ASRL3::INSTR"                                "COM3"              ""
5    "GPIB0::5::INSTR"                             "FGEN_2CH"          "Agilent Technolo
6    "GPIB0::11::INSTR"                            "OSCOPE_2CH"        "TEKTRONIX"
```

Create a `visadev` object using one of the resource names listed. If your instrument is not listed, it might not be configured properly in your VISA vendor's configuration utility software.

Note VISA-TCP/IP, VISA-Socket, and VISA-Serial instruments and devices might require additional configuration to appear in the `visadevlist` output.

---

5. You can create a VISA object to use with different instrument types. For example, create a VISA-Serial object connected to serial port `COM1`.

```
v = visadev("ASRL1::INSTR");
```

   If you do not get an error, the object was created successfully. If the resource name or alias does not exist, you will get an error. Check that the resource name or alias is correct in the vendor configuration utility software.

   You can have only one `visadev` object for a given resource at a time.

6. When you have connected, you can communicate with your device. If you have problems sending or receiving, you may need to configure communication settings. Make sure you are using the correct instrument command. Look in the instrument's documentation to see what commands it recognizes. Verify that communication works by testing the connection using the vendor's configuration utility.

VISA Driver Configuration

If you are still having connection or communication issues with your instrument using VISA, you can troubleshoot using your VISA vendor's software and utilities, as described in the following table.

VISA Vendor	Configuration Utility	Testing Connection	Debug Utility
Keysight VISA	Keysight Connection Expert (KCE)	Interactive IO button on KCE	IO Monitor button on KCE
NI-VISA	NI Measurement and Automation Explorer (NI MAX)	Tools > NI VISA > VISA Interactive Control	Tools > NI I/O Trace
Rohde & Schwarz R&S VISA	RsVisaConfigure, launched from RsVisa Config tab on RsVisaTester	RsVisaTester	RsVisaTraceTool, launched from RsVisa TraceTool tab on RsVisaTester

1. Use the VISA Conflict Manager settings from your VISA vendor's configuration utility to make sure that you have a preferred VISA set and that it is enabled. Check if all the VISA interfaces are using the expected VISA. For R&S VISA, make sure it is set to "Preferred". For example, for the Keysight Connection Expert, do the following.

   - Open the settings menu and select **Tools > VISA Conflict Manager**.

- Under **Enabled Implementations**, make sure that your VISA vendor is selected.
- Under **Preferred Implementation**, make sure that your VISA vendor is selected.

2. If you are using SCPI commands, check if your device responds to them as expected when issued from the configuration utility.
3. Use your VISA vendor's configuration utility to make sure that your device hardware is being detected. You can also check that your device responds to a `*IDN?` query.
4. Use your VISA vendor's debug utility to check the Instrument I/O traffic for errors other than timeout errors.
5. Try installing a different supported VISA vendor's driver.

See Also

`visadevlist` | `visadev`

Related Examples

- "Resolve VISA Connection Errors" on page 21-77
- "Resolve VISA Warning: Unable to Read Any Data" on page 21-68
- "Resolve VISA Warning: Unable to Read All Data" on page 21-70
- "Troubleshooting GPIB Interface" on page 21-19

Hardware Support Packages

The Instrument Control Toolbox includes support packages that allow the use of certain classes of instruments. To use any of the instrument support outlined below, install the appropriate support package. See the topics listed in the table for information about the contents and installing that support package.

Support Package	Contents and Installation
Aardvark I2C/SPI Interface	Use the I2C or SPI interface with the Aardvark adaptor. See “Install the Total Phase Aardvark I2C/SPI Interface Support Package” on page 15-9.
NI-845x I2C/SPI Interface	Use the I2C or SPI interface with the NI-845x adaptor. See “Install the NI-845x I2C/SPI Interface Support Package” on page 15-8.
NI-SCOPE Oscilloscopes	Use to communicate with NI-SCOPE oscilloscopes. Acquire waveform data from the oscilloscope and control it. See “Install the NI-SCOPE Oscilloscopes Support Package” on page 15-4.
NI-FGEN Function Generators	Use to communicate with NI-FGEN function generators. Control and configure the function generator, and perform tasks such as generating sine waves. See “Install the NI-FGEN Function Generators Support Package” on page 15-5.
NI-DCPower Power Supplies	Use to communicate with NI-DCPower power supplies. Control and take digital measurements from a power supply, such as the NI PXI 4011 triple-output programmable DC power supply. See “Install the NI-DCPower Power Supplies Support Package” on page 15-6.
NI-DMM Digital Multimeters	Use to communicate with NI-DMM digital multimeters. Control and take measurements from a digital multimeter, such as measuring voltage or resistance. See “Install the NI-DMM Digital Multimeters Support Package” on page 15-7.
NI-Switch Hardware	Use to communicate with NI-Switch instruments. For example, control a relay box such as the NI PXI-2586 10-channel relay switch. See “Install the NI-Switch Hardware Support Package” on page 15-10.

Support Package	Contents and Installation
NI VISA and ICP Interfaces	Use the Quick Control Oscilloscope and Quick Control Function Generator interfaces to communicate with and control oscilloscopes and function generators. See “Install the National Instruments VISA and ICP Interfaces Support Package” on page 15-11.

Deploying Standalone Applications with Instrument Control Toolbox

In this section...

"Tips for both interface based communication and driver-based communication" on page 21-38
"Tips for interface based communication" on page 21-38
"Tips for driver based communication" on page 21-38
"Hardware Support packages" on page 21-40

This topic contains tips for deploying standalone applications with MATLAB Compiler (deploytool or mcc) and Instrument Control Toolbox. Refer to these tips when creating standalone applications that use functionality from Instrument Control Toolbox.

Tips for both interface based communication and driver-based communication

- Device identifiers/resource names should not be hard-coded, as instrument resource names are likely to be different on other machines.
- A best practice is to use `instrhwinfo` and query the return output in your MATLAB code that you intend to deploy.

Tips for interface based communication

For direct interface based communication using I2C, SPI, GPIB, and VISA, on the deployment machine, install all required third-party drivers separately from the deployed application.

Tips for driver based communication

In addition to your MATLAB code, your deployed standalone application package should include files required by your application:

- Include MATLAB Instrument Driver (MDD file) in your standalone application project from `deploytool` or by passing a `-a` flag to `mcc` when compiling your MATLAB code.

IVI-C

Include the following in your deployed standalone application package:

- MATLAB Instrument Driver (MDD file)
- For 64-bit applications, MATLAB prototype and thunk files
- For 32-bit applications, MATLAB prototype file

The location of generated prototype and thunk files can be obtained from the result of executing:

```
sprintf('%s',[tempdir 'ICTDeploymentFiles'])
```

On the machine where you deploy your standalone application:

- All third-party drivers and dependencies need to be installed separately from the deployed standalone application.
- To reduce runtime unknowns, install the same version of IVI-C driver on the deployment system as used on the development system.
- To reduce runtime unknowns, install the same version of VISA driver libraries on the deployment system as used on the development system.

Note To troubleshoot vendor driver installation issues, it is recommended that the deployed application provide a way to simulate connection to the hardware by instantiating the driver with `'optionstring', 'simulate=true'` as arguments for `icdevice`. This will help narrow down the root cause of deployment issues to vendor driver installation issue, or hardware issues.

Quick Control Interfaces

If you are not using the default SCPI-based drivers for Quick Control Oscilloscope (`'tektronix'`) and Quick Control Function Generator (`'Agilent332x0_SCPI'`), and are instead using an IVI-C driver, include the following in your deployed standalone application package:

- For 64-bit applications, MATLAB prototype and thunk files for IVIScope or IVIFGen
- For 32-bit applications, MATLAB prototype file for IVIScope or IVIFGen

The location of generated prototype and thunk files can be obtained from the result of executing:

```
sprintf('%s',[tempdir 'ICTDeploymentFiles'])
```

On the machine where you deploy your standalone application:

- If required, all third-party drivers and dependencies need to be installed separately from the deployed standalone application.
- To reduce runtime unknowns, install the same version of IVI-C driver on the deployment system as used on the development system.
- To reduce runtime unknowns, install the same version of VISA driver libraries on the deployment system as used on the development system.

Generic MDD

For use with the generic MDD, include the following in your deployed standalone application package:

- MATLAB Instrument Driver (MDD file)

If your MDD uses LOADLIBRARY to interface with a C shared library, include:

- For 64-bit applications, MATLAB prototype and thunk files for the C shared library
- For 32-bit applications, MATLAB prototype file for the C shared library
- MATLAB prototype and thunk files for a C shared library can be generated on a development machine (with a supported C compiler) by using LOADLIBRARY command.

On the machine where you deploy your standalone application:

- If required, all third-party drivers and dependencies need to be installed separately from the deployed standalone application.

Hardware Support packages

For more info on deploying standalone applications which use functionality installed as a support package:

```
web(fullfile(docroot, 'compiler/manage-support-packages.html'))
```

Contact MathWorks and Use the instrsupport Function

If you need support from MathWorks, visit our Web site at `https://www.mathworks.com/support/`.

Before contacting MathWorks, run the `instrsupport` function. This function returns diagnostic information such as:

- The versions of MathWorks products you are using
- Your MATLAB path
- The characteristics of your hardware
- Information about your adaptors

The output from `instrsupport` is automatically saved to a text file, `instrsupport.txt`, which you can use to help troubleshoot your problem.

To have MATLAB generate this file for you, type

```
instrsupport
```

Serial Warning - Unable to Read Any Data

These remedies apply to the case when you receive no data and you get this warning message:

```
'serial' unable to read any data
```

When using the Serial interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions
- Reading binary data using the `fread` function
- Reading binblock data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
An invalid command was sent to the device, so there is a problem reading the response to the command.	Check your device manual for proper command formatting.
Your device is connected to an incorrect serial port.	Verify that your device is connected to the specified port. It must match the port you specify when you create the `serial` object. For information about specifying the port, see `serial`.
An incorrect write terminator was sent to the instrument before attempting to read data, so there is no data to read.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see Terminator.
Your device is not configured to send data on the serial port.	Verify the device communication settings. For more information about communication settings, see “Create Serial Port Object” on page 6-13 and “Configure Serial Port Communication Settings” on page 6-15.

More Troubleshooting Help

For more information about troubleshooting the Serial interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see “Troubleshooting Serial Port Interface” on page 21-16.

Serial Warning - Unable to Read All Data

These remedies apply to the case when you receive some data and you get this warning message:

```
'serial' unable to read all requested data
```

ASCII Data

When using the Serial interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions

these are possible causes and remedies:

Cause	Solution
An incorrect read terminator was used.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see Terminator.
Communication with the device was interrupted.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting Serial Port Interface” on page 21-16.
There is an invalid serial port configuration.	Verify that the `serial` object properties match the device serial settings. For more information about configuring settings, see “Create Serial Port Object” on page 6-13 and “Configure Serial Port Communication Settings” on page 6-15.

Binary Data

When using the Serial interface for:

- Reading binary data using the `fread` function

these are possible causes and remedies:

Cause	Solution
The number of values to read was not specified and was set to the `InputBufferSize` by default.	Set the number of values to read using the `size` argument on the `fread` function, or change the `InputBufferSize` property. For more information about setting the property, see InputBufferSize. For information about setting the number of values to read, see `fread`.
Device did not send all the requested data.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting Serial Port Interface” on page 21-16.
There was a data format mismatch.	Verify that the device data format matches the specified read format. Data format is set using the `precision` property. For more information about supported precisions, see `fread`.

Binblock Data

When using the Serial interface for:

- Reading binblock (binary-block) data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
The timeout value might be too short for the amount of data being read.	Increase the `Timeout` property value. For more information about setting the property, see Timeout.
Communication with the device was interrupted.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting Serial Port Interface” on page 21-16.

More Troubleshooting Help

For more information about troubleshooting the Serial interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see “Troubleshooting Serial Port Interface” on page 21-16.

TCP/IP Warning - Unable to Read Any Data

These remedies apply to the case when you receive no data and you get this warning message:

```
'tcpip' unable to read any data
```

When using the TCP/IP interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions
- Reading binary data using the `fread` function
- Reading binblock data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
An invalid command was sent to the device, so there is a problem reading the response to the command.	Check your device manual for proper command formatting.
An incorrect write terminator was sent to the instrument before attempting to read data, so there is no data to read.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see Terminator.
Your device did not receive the command because the remote host address or the remote port is incorrect.	Verify that the device is at the remote host address that you specified, and is listening on the remote port that you specified when you created the `tcpip` object.

TCP/IP Warning - Unable to Read All Data

These remedies apply to the case when you receive some data and you get this warning message:

```
'tcpip' unable to read all requested data
```

ASCII Data

When using the TCP/IP interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions

these are possible causes and remedies:

Cause	Solution
An incorrect read terminator was used.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see Terminator.
Communication with the device was interrupted.	Check your device connection.

Binary Data

When using the TCP/IP interface for:

- Reading binary data using the `fread` function

these are possible causes and remedies:

Cause	Solution
The number of values to read was not specified and was set to the `InputBufferSize` by default.	Set the number of values to read using the `size` argument on the `fread` function, or change the `InputBufferSize` property. For more information about setting the property, see InputBufferSize. For information about setting number of values to read, see `fread`.
Device did not send all the requested data.	Check your device connection.
There was a data format mismatch.	Verify that the device data format matches the specified read format. Data format is set using the `Precision` property. For more information about supported precisions, see `fread`.

Binblock Data

When using the TCP/IP interface for:

- Reading binblock (binary-block) data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
The timeout value might be too short for the amount of data being read.	Increase the `Timeout` property value. For more information about setting the property, see Timeout.
Communication with the device was interrupted.	Check your device connection.

UDP Warning - Unable to Read Any Data

These remedies apply to the case when you receive no data and you get this warning message:

```
'udp' unable to read any data
```

ASCII and Binary Data

When using the UDP interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions
- Reading binary data using the `fread` function

these are possible causes and remedies:

Cause	Solution
An invalid command was sent to the device, so there is a problem reading the response to the command.	Check your device manual for proper command formatting.
The device did not receive the command because of an incorrect UDP `RemotePort` value.	Verify that the UDP `RemotePort` value is set to the port number the device is listening on. For more information about setting the remote port, see `udp` and "Create a UDP Object and View Properties" on page 8-2.
A firewall is blocking incoming UDP packets.	Verify that your system firewall setting allows connections to 'LocalPort'.
The UDP packet size is larger than the maximum packet size that can be handled by the Ethernet adaptor.	The UDP packet size is controlled by the `OutputDatagramPacketSize` property. You can specify the size, in bytes, between 1 and 65,535, and the default value is 512.
You might need to enable port sharing.	If you are receiving UDP broadcasts on a shared port, set the `EnablePortSharing` property to `on`. For the syntax, see "Enable Port Sharing Over UDP" in "Create a UDP Object and View Properties" on page 8-2.

Binblock Data

When using the UDP interface for:

- Reading binblock (binary-block) data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
An incorrect write terminator was sent to the instrument before attempting to read data, so there is no data to read.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see Terminator.
An invalid command was sent to the device, so there is a problem reading the response to the command.	Check your device manual for proper command formatting.

Cause	Solution
The device did not receive the command because of an incorrect UDP `RemotePort` value.	Verify that the UDP `RemotePort` value is set to the port number the device is listening on. For more information about setting the remote port, see `udp` and “Create a UDP Object and View Properties” on page 8-2.
A firewall is blocking incoming UDP packets.	Verify that your system firewall setting allows connections to 'LocalPort'.
The UDP packet size is larger than the maximum packet size that can be handled by the Ethernet adaptor.	The UDP packet size is controlled by the `OutputDatagramPacketSize` property. You can specify the size, in bytes, between 1 and 65,535, and the default value is 512.
You might need to enable port sharing.	If you are receiving UDP broadcasts on a shared port, set the `EnablePortSharing` property to `on`. For the syntax, see "Enable Port Sharing Over UDP" in “Create a UDP Object and View Properties” on page 8-2.
If the amount of data being received spans multiple UDP packets, it is possible that the system dropped packets.	The UDP packet size is controlled by the `OutputDatagramPacketSize` property. You can specify the size, in bytes, between 1 and 65,535, and the default value is 512. You can increase or decrease the packet size if necessary.

More Troubleshooting Help

For more information about troubleshooting the UDP interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see “Troubleshooting UDP Interface” on page 21-27.

UDP Warning - Unable to Read All Data

These remedies apply to the case when you receive some data and you get this warning message:

```
'udp' unable to read all requested data
```

ASCII Data

When using the UDP interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions

these are possible causes and remedies:

Cause	Solution
An incorrect read terminator was used.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see Terminator.
Communication with the device was interrupted.	Check your device connection. For more information about troubleshooting configuration and connection, see "Troubleshooting UDP Interface" on page 21-27.

Binary Data

When using the UDP interface for:

- Reading binary data using the `fread` function

these are possible causes and remedies:

Cause	Solution
The number of values to read was not specified and was set to the `InputBufferSize` by default.	Set the number of values to read using the `size` argument on the `fread` function, or change the `InputBufferSize` property. For more information about setting the property, see InputBufferSize. For information about setting number of values to read, see `fread`.
Device did not send all the requested data.	Check your device connection. For more information about troubleshooting configuration and connection, see "Troubleshooting UDP Interface" on page 21-27.
There was a data format mismatch.	Verify that the device data format matches the specified read format. Data format is set using the `Precision` property. For more information about supported precisions, see `fread`.
If the amount of data being received spans multiple UDP packets, it is possible that the system dropped packets.	The UDP packet size is controlled by the `OutputDatagramPacketSize` property. You can specify the size, in bytes, between 1 and 65,535, and the default value is 512. You can increase or decrease the packet size if necessary.

Binblock Data

When using the UDP interface for:

- Reading binblock (binary-block) data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
The timeout value might be too short for the amount of data being read.	Increase the `Timeout` property value. For more information about setting the property, see Timeout.
Communication with the device was interrupted.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting UDP Interface” on page 21-27.

More Troubleshooting Help

For more information about troubleshooting the UDP interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see “Troubleshooting UDP Interface” on page 21-27.

GPIB Warning - Unable to Read Any Data

These remedies apply to the case when you receive no data and you get this warning message:

```
'gpib' unable to read any data
```

When using the GPIB interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions
- Reading binary data using the `fread` function
- Reading binblock data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
The device did not receive the command because of an incorrect GPIB address or resource string.	Verify that the device is at the GPIB address that you specified when you created the `gpib` object. For information about setting the address, see `gpib`.
The `EOIMode` property is not set correctly.	Verify that the `EOIMode` property is set to the value required by your device. You can set it to `on` or `off`, and `on` is the default. For more information about setting the property, see “Write and Read GPIB Data” on page 4-10.
You might have connected to the wrong instrument.	Check the GPIB address. Verify that the device is at the GPIB address that you specified when you created the `gpib` object. For information about setting the address, see `gpib`.

More Troubleshooting Help

For more information about troubleshooting the GPIB interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see “Troubleshooting GPIB Interface” on page 21-19.

GPIB Warning - Unable to Read All Data

These remedies apply to the case when you receive some data and you get this warning message:

```
'gpib' unable to read all requested data
```

ASCII Data

When using the GPIB interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions

these are possible causes and remedies:

Cause	Solution
The `EOIMode` property is not set correctly.	Verify that the `EOIMode` property is set to the value required by your device. You can set it to `on` or `off`, and `on` is the default. For more information about setting the property, see “Write and Read GPIB Data” on page 4-10 .
The `EOIMode` property is set to `off`, but the `EOSMode` and `EOSCharCode` properties might not be configured to the same setting as the instrument.	Verify that the `EOSMode` and `EOSCharCode` properties are configured to the same settings as your device. For more information about setting the properties, see EOSMode and EOSCharCode.
Communication with the device was interrupted.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting GPIB Interface” on page 21-19.

Binary Data

When using the GPIB interface for:

- Reading binary data using the `fread` function

these are possible causes and remedies:

Cause	Solution
Device did not send all the requested data.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting GPIB Interface” on page 21-19.
There was a data format mismatch.	Verify that the device data format matches the specified read format. Data format is set using the `Precision` property. For more information about supported precisions, see `fread`.
The number of values to read was not specified and was set to the `InputBufferSize` by default.	Set the number of values to read using the `size` argument on the `fread` function, or change the `InputBufferSize` property. For more information about setting the property, see InputBufferSize. For information about setting number of values to read, see `fread`.

Binblock Data

When using the GPIB interface for:

- Reading binblock (binary-block) data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
The timeout value might be too short for the amount of data being read.	Increase the `Timeout` property value. For more information about setting the property, see Timeout.
Communication with the device was interrupted.	Check your device connection. For more information about troubleshooting configuration and connection, see "Troubleshooting GPIB Interface" on page 21-19.

More Troubleshooting Help

For more information about troubleshooting the GPIB interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see "Troubleshooting GPIB Interface" on page 21-19.

TCP/IP Socket Using VISA Warning - Unable to Read Any Data

These remedies apply to the case when you receive no data and you get this warning message:

```
'visa' unable to read any data
```

When using the VISA TCP/IP Socket for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions
- Reading binary data using the `fread` function
- Reading binblock data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
An invalid command was sent to the device, so there is a problem reading the response to the command.	Check your device manual for proper command formatting.
The write `EOSCharCode` is incorrect.	Verify that the `EOSCharCode` property is set to the value required by your device. For more information about setting the property, see EOSCharCode.
Your device did not receive the command because either the TCP/IP remote host address or the remote port is incorrect.	Verify that the device is at the remote host address that you specified, and is listening on the remote port that you specified when you created the `visa` object. For more information about communication settings, see "Create TCP/IP Client and Configure Settings" on page 7-3 and "Write and Read Data over TCP/IP Interface" on page 7-7.

More Troubleshooting Help

For more information about troubleshooting the TCP/IP Socket using VISA interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see "Troubleshooting TCP/IP Client Interface" on page 21-23 and "Troubleshooting IVI and Quick-Control Interfaces" on page 21-29.

TCP/IP Socket Using VISA Warning - Unable to Read All Data

These remedies apply to the case when you receive some data and you get this warning message:

```
'visa' unable to read all requested data
```

ASCII Data

When using the VISA TCP/IP Socket for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions

these are possible causes and remedies:

Cause	Solution
The read `EOSCharCode` is incorrect.	Verify that the `EOSCharCode` property is set to the value required by your device. For more information about setting the property, see EOSCharCode.
Communication with the device was interrupted.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting TCP/IP Client Interface” on page 21-23 and “Troubleshooting IVI and Quick-Control Interfaces” on page 21-29.

Binary Data

When using the VISA TCP/IP Socket for:

- Reading binary data using the `fread` function

these are possible causes and remedies:

Cause	Solution
The number of values to read was not specified and was set to the `InputBufferSize` by default.	Set the number of values to read using the `Count` property on the `fread` function, or change the `InputBufferSize` property. For more information about setting the property, see InputBufferSize. For information about setting number of values to read, see `fread`.
Device did not send all the requested data.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting TCP/IP Client Interface” on page 21-23 and “Troubleshooting IVI and Quick-Control Interfaces” on page 21-29.
There was a data format mismatch.	Verify that the device data format matches the specified read format. Data format is set using the `Precision` property. For more information about supported precisions, see `fread`.

Binblock Data

When using the VISA TCP/IP Socket for:

- Reading binblock (binary-block) data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
The timeout value might be too short for the amount of data being read.	Increase the `Timeout` property value. For more information about setting the property, see Timeout.
Communication with the device was interrupted.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting TCP/IP Client Interface” on page 21-23 and “Troubleshooting IVI and Quick-Control Interfaces” on page 21-29.

More Troubleshooting Help

For more information about troubleshooting the TCP/IP Socket using VISA interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see “Troubleshooting TCP/IP Client Interface” on page 21-23 and “Troubleshooting IVI and Quick-Control Interfaces” on page 21-29.

Bluetooth Warning - Unable to Read Any Data

These remedies apply to the case when you receive no data and you get this warning message:

```
'Bluetooth' unable to read any data
```

When using the Bluetooth interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions
- Reading binary data using the `fread` function
- Reading binblock data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
An invalid command was sent to the device, so there is a problem reading the response to the command.	Check your device manual for proper command formatting.
An incorrect write terminator was sent to the instrument before attempting to read data, so there is no data to read.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see Terminator.
An incorrect `RemoteName`, `RemoteID`, or `Channel` was used.	Verify that the device is paired and connected.

Bluetooth Warning - Unable to Read All Data

These remedies apply to the case when you receive some data and you get this warning message:

```
'Bluetooth' unable to read all requested data
```

ASCII Data

When using the Bluetooth interface for:

- Reading ASCII (text) data using the `fscanf`, `fgets`, or `fgetl` functions

these are possible causes and remedies:

Cause	Solution
An incorrect read terminator was used.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see Terminator.
Communication with the device was interrupted.	Check your device connection.

Binary Data

When using the Bluetooth interface for:

- Reading binary data using the `fread` function

these are possible causes and remedies:

Cause	Solution
The number of values to read was not specified and was set to the `InputBufferSize` by default, or the number of values to be read was specified to a value greater than the `BytesAvailable` property.	Set the number of values to read using the `size` argument on the `fread` function, or change the `InputBufferSize` property. For more information about setting the property, see InputBufferSize. For information about setting number of values to read, see `fread`.
Device did not send all the requested data.	Check your device connection.
There was a data format mismatch.	Verify that the device data format matches the specified read format. Data format is set using the `Precision` property. For more information about supported precisions, see `fread`.

Binblock Data

When using the Bluetooth interface for:

- Reading binblock (binary-block) data using the `binblockread` function

these are possible causes and remedies:

Cause	Solution
The timeout value might be too short for the amount of data being read.	Increase the `Timeout` property value. For more information about setting the property, see Timeout.
Communication with the device was interrupted.	Check your device connection.

Serialport Warning - Unable to Read Any Data

These remedies apply to the case when you receive no data and you get this warning message:

```
'serialport' unable to read any data.
```

When using the serialport interface for:

- Reading ASCII (text) data using the `readline` or `writeread` functions
- Reading binary data using the `read` function
- Reading binblock data using the `readbinblock` function

These are possible causes and remedies:

Cause	Solution
An invalid command was sent to the device, so there is a problem reading the response to the command.	Check your device manual for proper command formatting.
Your device is connected to an incorrect serial port.	Verify that your device is connected to the specified port. It must match the port you specify when you create the `serialport` object. For information about specifying the port, see `serialport`.
An incorrect write terminator was sent to the instrument before attempting to read data, so there is no data to read.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see `configureTerminator`.
Your device is not configured to send data on the serial port.	Verify the device communication settings. For more information about communication settings, see "Create Serial Port Object" on page 6-13 and "Configure Serial Port Communication Settings" on page 6-15.

More Troubleshooting Help

For more information about troubleshooting the Serial interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see "Troubleshooting Serial Port Interface" on page 21-16.

Serialport Warning - Unable to Read All Data

These remedies apply to the case when you receive some data and you get this warning message:

```
'serialport' unable to read all requested data.
```

When using the serialport interface for:

- Reading binary data using the `read` function

These are possible causes and remedies:

Cause	Solution
The number of values to read was set to a higher value than what was available to be read.	Set the number of values to read using the input argument `count` on the `read` function. For information about setting the number of values to read, see `read`.
Device did not send all the requested data.	Check your device connection. For more information about troubleshooting configuration and connection, see "Troubleshooting Serial Port Interface" on page 21-16.
There was a data format mismatch.	Verify that the device data format matches the specified read format. Data format is set using the input argument `precision` on the `read` function. For more information about supported precisions, see `read`.

More Troubleshooting Help

For more information about troubleshooting the Serial interface, including supported platforms, adaptor requirements, configuration and connection, and other troubleshooting tips, see "Troubleshooting Serial Port Interface" on page 21-16.

Resolve TCP/IP Client Warning: Unable to Read Any Data

Issue

These remedies apply when you receive no data and you get this warning message:

```
'tcplient' unable to read any data
```

Possible Solutions

Try these remedies to resolve the following causes for when you use the TCP/IP client interface for:

- Reading ASCII (text) data using the `readline` function.
- Reading binary data using the `read` function.
- Reading binblock data using the `readbinblock` function.

Cause	Solution
An invalid command was sent to the device, so there is a problem reading the response to the command.	Check your device manual for proper command formatting.
An incorrect write terminator was sent to the instrument before attempting to read data, so there is no data to read.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see `configureTerminator`.
Your device did not receive the command because the remote host address or the remote port is incorrect.	Verify that the device is at the remote host address that you specified, and is listening on the remote port that you specified when you created the `tcpclient` object. For more information about communication settings, see “Create TCP/IP Client and Configure Settings” on page 7-3 and “Write and Read Data over TCP/IP Interface” on page 7-7.

See Also

`configureTerminator` | `read` | `readline`

More About

- “Troubleshooting TCP/IP Client Interface” on page 21-23

Resolve TCP/IP Server Warning: Unable to Read Any Data

Issue

These remedies apply when you receive no data and you get this warning message:

```
'tcpserver' unable to read any data
```

Possible Solutions

Try these remedies to resolve the following causes for when you use the TCP/IP server interface for:

- Reading ASCII (text) data using the `readline` function.
- Reading binary data using the `read` function.
- Reading binblock data using the `readbinblock` function.

Cause	Solution
An invalid command was sent to the connected client, so there is a problem reading the response to the command from the server.	Check your client's documentation for proper command formatting.
An incorrect write terminator was sent to the connected client before attempting to read data, so there is no data to read.	Verify that the server's `Terminator` property is set to the value required by your client. The server and its connected client must have the same terminator. For more information about setting the property, see `configureTerminator`.
The server did not receive the data because a client was not connected to it or the connected client did not send any data.	Verify that the client is created using the address and port that you specified when you created the `tcpserver` object. Check that the value of the `Connected` property is `1` (`true`) for the server. Also verify that the connected client has attempted to send data to the server.

See Also

`read` | `readline` | `readbinblock`

Related Examples

- "Troubleshooting TCP/IP Server Interface" on page 21-25

Resolve TCP/IP Server Warning: Unable to Read All Data

Issue

These remedies apply when you receive some data and you get this warning message:

```
'tcpserver' unable to read all requested data
```

Possible Solutions

Try these remedies to resolve the following causes when you use the TCP/IP server interface for reading binary data using the `read` function.

Cause	Solution
The number of values to read was set to a higher value than the number of values available to be read.	Confirm that you set the correct number of values to read with the `read` function. Check the amount of data available with the `NumBytesAvailable` property of the `tcpserver` object. For information about setting the number of values to read, see `read`.
Client did not send all the requested data.	Check your client connection. For more information about troubleshooting configuration and connection, see “Troubleshooting TCP/IP Server Interface” on page 21-25.
There was a data format mismatch.	Verify that the client data type matches the specified server read data type. Specify the server data type with `read`. For information about supported data types, see `read`.

See Also

`read`

Related Examples

- “Troubleshooting TCP/IP Server Interface” on page 21-25

Resolve UDP Port Warning: Unable to Read Any Data

Issue

These remedies apply when you receive no data and you get this warning message:

```
'udpport' unable to read any data
```

Possible Solutions

Try these remedies to resolve the following causes when you use the UDP interface for:

- Reading ASCII (text) data using the `readline` function.
- Reading binary data using the `read` function.

Cause	Solution
An invalid command was sent to the destination UDP port, so there is a problem reading the response to the command.	Check the manual of the destination device for proper command formatting.
An incorrect write terminator was sent to the destination UDP port before attempting to read data, so there is no data to read.	Verify that the `Terminator` property is set to the value required by your destination UDP port. For more information about setting the property, see `configureTerminator`.
The destination UDP port did not receive the command because of an incorrect destination port value.	Verify that the UDP destination port value is set to the port number the device is listening on. For more information about setting the destination port, see `write` or `writeline`.
A firewall is blocking incoming UDP packets.	Verify that your system firewall setting allows connections to `LocalPort`.
The UDP packet size is larger than the maximum packet size that can be handled by the Ethernet adaptor.	The UDP packet size is controlled by the `OutputDatagramSize` property. You can specify the size, in bytes, between 1 and 65507, and the default value is 512.

See Also

More About

- "Troubleshooting UDP Interface" on page 21-27

Resolve UDP Port Warning: Unable to Read All Data

Issue

These remedies apply when you receive some data and you get this warning message:

```
'udpport' unable to read all requested data
```

Possible Solutions

Try these remedies to resolve the following causes when you use the UDP interface for reading binary data using the `read` function.

Cause	Solution
The number of values to read was larger than the number of values available.	Confirm that you have the correct number of values of data available before you read. Check the amount of data available with the `NumBytesAvailable` or `NumDatagramsAvailable` property of the `udpport` object.
Device did not send all the requested data.	Check your device connection. For more information about troubleshooting configuration and connection, see “Troubleshooting UDP Interface” on page 21-27.
There was a data format mismatch.	Verify that the device data type matches the specified read data type. Specify the data type with `read`. For more information about supported precisions, see `read`.

See Also

More About

- “Troubleshooting UDP Interface” on page 21-27

Resolve VISA Warning: Unable to Read Any Data

Issue

These remedies apply when you receive no data and you get this warning message:

```
'visadev' unable to read any data
```

Possible Solutions

Try these remedies to resolve the following causes for when you use the VISA interface for:

- Reading ASCII (text) data using the `readline` function.
- Reading binary data using the `read` function.
- Reading binblock data using the `readbinblock` function.

Cause	Solution
The device did not receive the command because of an incorrect resource name. You might have connected to the wrong device.	Verify that the device is associated with the resource name that you specified when you created the `visadev` object.
An invalid command was sent to the device, so there is a problem reading the response to the command.	Check your device documentation for proper command formatting.
An incorrect write terminator was configured before attempting to read data, so there is no data to read.	Verify that the `Terminator` property is set to the value required by your device. For more information about setting the property, see `configureTerminator`.
The device is not configured to read binblock data.	Configure the instrument for binblock data. Check your device documentation.

Try these steps for the VISA-Serial interface:

Cause	Solution
Your device is connected to an incorrect serial port.	Verify that your device is connected to the specified port. It must match the port you specify when you create the `visadev` object.
Your device is not configured to send data on the serial port.	Verify the device communication settings.

Try these steps for the VISA-Socket interface:

Cause	Solution
Your device did not receive the command because either the TCP/IP remote host address or the remote port is incorrect.	Verify that the device is at the remote host address that you specified, and is listening on the remote port that you specified when you created the `visadev` object.

See Also

`read` | `readline` | `readbinblock`

Related Examples

- "Troubleshooting VISA Interface" on page 21-33

Resolve VISA Warning: Unable to Read All Data

Issue

These remedies apply when you receive some data and you get this warning message:

```
'visadev' unable to read all requested data
```

Possible Solutions

Try these remedies to resolve the following causes when you use the VISA interface for reading binary data using the `read` function.

Cause	Solution
The number of values to read was set to a higher value than the number of values available to be read.	Confirm that you set the correct number of values to read with the `read` function. Check the amount of data available with the `NumBytesAvailable` property of the `visadev` object. For information about setting the number of values to read, see `read`.
Device did not send all the requested data or communication with the device was interrupted.	Check your device connection. For more information about troubleshooting configuration and connection, see "Troubleshooting VISA Interface" on page 21-33.
There was a data format mismatch.	Verify that the device data type matches the specified read data type. Specify the data type with `read`. For more information about supported data types, see `read`.
The timeout value might be too short for the amount of data being read.	Increase the `Timeout` property value. For more information about setting the property, see "Timeout" on page 25-0 .

See Also

`read`

Related Examples

- "Troubleshooting VISA Interface" on page 21-33

Resolve Serial Port Connection Errors

Issue

If you are unable to connect to a serial port device using the `serialport` interface, follow these troubleshooting steps.

Possible Solutions

Check Device Status

Check that the specified port is not in use.

- Make sure that a `serialport` object using the same port number does not already exist in the workspace. You can create only one `serialport` object for each port.
- Check that your device is not in use outside of MATLAB. Disconnect your device from any other devices, applications, or programs.
- Use a third-party serial communication software, such as PuTTY, to check that you can access the specified port from other software.
- Make sure that your device is powered on and connected to your computer.

Verify Port Name

Check that the specified port name is correct and that a device is connected to it.

- Use the `serialportlist` function to return a list of all serial ports that you have access to on your computer. Use `serialportlist("available")` to return a list of only serial ports that are available. Make sure that you are creating a `serialport` object using one of the listed ports.
- Check from your computer settings that the device connected to the serial port is available. For more information on how to view this information on your platform, see “Find Serial Port Information for Your Platform” on page 6-9.

Specify Supported Parameters

Check that the specified parameters are supported by your device.

- Make sure that the baud rate specified as an input argument is supported by your device. Refer to your device documentation for this information. The baud rate must match the device configuration.
- If you have specified any other parameters using name-value arguments, make sure that those are supported by your device as well. You can specify the `DataBits`, `Parity`, `StopBits`, `FlowControl`, `ByteOrder`, and `Timeout` properties using name-value arguments.

Additional Troubleshooting for Virtual Serial Port Connection

If you have devices that present themselves as serial ports on your operating system, you can use them as virtual USB serial ports. One example of such a device is a USB serial dongles.

- If you are connecting to a device over a virtual serial port, check that the device drivers are properly installed.

- If you are using Linux, you might need to enable permissions to read from and write to a virtual serial port. Some Linux distributions require the user account to be a member of the dialout group to have permission to read from and write to the serial port.

See Also

`serialport`

Related Examples

- "Troubleshooting Serial Port Interface" on page 21-16

Resolve TCP/IP Client Connection Errors

Issue

If you are unable to create a TCP/IP client using the `tcpclient` interface, follow these troubleshooting steps.

Possible Solutions

Check IP Address and Port

- If your instrument supports TCP socket communication by allowing control and communication on a static TCP/IP port, the port number is typically set by the manufacturer. However, you might need to enable a TCP socket server on the instrument. Run the following commands in MATLAB with your remote host IP address or host name and port. In this example, the remote host IP address is `192.168.1.111` and port is `4000`.

  1. Check that the IP address or host name you specified can be reached.

     ```
     !ping 192.168.1.111
     ```
  2. Check that the port is accessible using Telnet or another application that supports Telnet, such as PuTTY.

     ```
     !telnet 192.168.1.111 4000
     ```
- Verify that the specified IP address or host name is valid by using `resolvehost`. If the output is empty, the specified IP address or host name is invalid. For example, if the host name is `en.wikipedia.org`, run the following.

  ```
  [name,address] = resolvehost("en.wikipedia.org")
  ```

  ```
  name =

      'en.wikipedia.org'


  address =

      '208.80.154.224'
  ```
- Make sure that you specified the correct port number. Refer to your instrument manual or contact your instrument manufacturer to check the port number.
- Check that a TCP socket server is listening at the specified port. For example, on certain Tektronix oscilloscope models, you might have to enable the socket server, which is a feature provided by TekVISA running on the instrument itself and can provide a raw TCP connection for instrument control.
- Use a third-party TCP/IP communication software, such as PuTTY, to check that you can access the specified IP address and port from other software.

Check Network Connection

Make sure that your network connection is configured properly.

- Make sure that your network adapter is enabled and connected.
- If the instrument has a built-in PC running the TCP socket server, check if a firewall on the instrument's PC is blocking communication.
- Check if a firewall on the client computer is blocking communication.
- Some instruments and servers (such as `tcpserver` objects) allow only one connection. Make sure that the instrument or server is not already connected to another client.

See Also

`tcpclient`

Related Examples

- "Troubleshooting TCP/IP Client Interface" on page 21-23

Resolve TCP/IP Server Connection Errors

Issue

If you are unable to create a TCP/IP server using the `tcpserver` interface, follow these troubleshooting steps.

Possible Solutions

- Make sure that your network adapter is enabled and connected.
- Check that the IP address you specify is available on your machine. To see valid IP addresses for your machine, run the following command in MATLAB on Windows.

  ```
  !ipconfig
  ```

  On Linux and macOS, run the following command.

  ```
  !ifconfig
  ```
- Verify that the specified IP address or host name is valid by using `resolvehost`. If the output is empty, the specified IP address or host name is invalid. For example, if the host name is `en.wikipedia.org`, run the following.

  ```
  [name,address] = resolvehost("en.wikipedia.org")
  ```

  ```
  name =

      'en.wikipedia.org'


  address =

      '208.80.154.224'
  ```
- Make sure that you do not specify a port that is already in use. In addition, you can only create one server object for a given address and port combination. To see all TCP/IP ports in use, run the following command in MATLAB.

  ```
  !netstat -a -n -p TCP
  ```

See Also

`tcpserver`

Related Examples

- “Troubleshooting TCP/IP Server Interface” on page 21-25

Resolve UDP Port Connection Errors

Issue

If you are unable to connect to a UDP socket using the `udpport` interface, follow these troubleshooting steps.

Possible Solutions

- Make sure that your network adapter is enabled and connected.
- Make sure that you do not specify a local port that is already in use. In addition, you can only create one `udpport` object for a given host and port combination. To see all local ports in use, run the following command in MATLAB.

  ```
  !netstat -a -n -p UDP
  ```
- If you are using the same port for multiple UDP sockets, make sure that port sharing is enabled for the specified port.
- If you specified `LocalHost` as a name-value argument, make sure that it represents your computer's host name or IP address that you want to listen on for incoming UDP packets. If you want to specify the host name or address to send packets to, specify it in the `write` (byte-type and datagram-type `udpport` objects) or `writeline` (byte-type `udpport` objects) functions.
- If you specified a local host, check that the local host IP address is available on your machine. To see valid IP addresses for your machine, run the following command in MATLAB on Windows.

  ```
  !ipconfig
  ```

  On Linux and macOS, run the following command.

  ```
  !ifconfig
  ```
- Verify that the specified local host name is valid by using `resolvehost`.

See Also

`udpport`

Related Examples

- “Troubleshooting UDP Interface” on page 21-27

Resolve VISA Connection Errors

Issue

If you are unable to connect to a VISA device using the `visadev` interface, follow these troubleshooting steps.

Possible Solutions

Configuration and Connection

1. Make sure your device is powered on and all cables are properly connected.
2. Make sure that you have the correct instrument driver installed for your device. Refer to your device documentation and the vendor website.

> **Note** If you are connecting to a GPIB device using an NI GPIB adaptor, you must download the NI-488.2 driver compatible with your VISA driver version from the NI website. The NI-488.2 driver is not available as an Instrument Control Toolbox support package.

3. Make sure that your device is supported in Instrument Control Toolbox. See “Is My Hardware Supported?” on page 21-4 and “Instrument Control Toolbox Supported Hardware”.
4. Make sure that Instrument Control Toolbox recognizes your device, by using the `visadevlist` function.

```
resourceList = visadevlist
```

```
resourceList = 

  6×6 table

                     ResourceName                           Alias                Vendor
         _______________________________________________    __________________    __________________

    1    "USB0::0x0699::0x036A::CU010105::0::INSTR"    "NI_SCOPE_4CH"        "TEKTRONIX"
    2    "TCPIP0::169.254.2.20::inst0::INSTR"          "Keysight_33210A"     "Agilent Technolo
    3    "ASRL1::INSTR"                                "COM1"                ""
    4    "ASRL3::INSTR"                                "COM3"                ""
    5    "GPIB0::5::INSTR"                             "FGEN_2CH"            "Agilent Technolo
    6    "GPIB0::11::INSTR"                            "OSCOPE_2CH"          "TEKTRONIX"
```

Create a `visadev` object using one of the resource names listed. If your instrument is not listed, it might not be configured properly in your VISA vendor’s configuration utility software.

> **Note** VISA-TCP/IP, VISA-Socket, and VISA-Serial instruments and devices might need to be added manually in the VISA vendor's configuration utility to appear in the `visadevlist` output.

VISA Driver Configuration

If you are still having connection or communication issues with your instrument using VISA, you can troubleshoot using your VISA vendor's software and utilities, as described in the following table.

VISA Vendor	Configuration Utility	Testing Connection	Debug Utility
Keysight VISA	Keysight Connection Expert (KCE)	Interactive IO button on KCE	IO Monitor button on KCE
NI-VISA	NI Measurement and Automation Explorer (NI MAX)	Tools > NI VISA > VISA Interactive Control	Tools > NI I/O Trace
Rohde & Schwarz R&S VISA	RsVisaConfigure, launched from RsVisa Config tab on RsVisaTester	RsVisaTester	RsVisaTraceTool, launched from RsVisa TraceTool tab on RsVisaTester

1. Use the VISA Conflict Manager settings from your VISA vendor's configuration utility to make sure that you have a preferred VISA set and that it is enabled. Check if all the VISA interfaces are using the expected VISA. For R&S VISA, make sure it is set to "Preferred". For example, for the Keysight Connection Expert, do the following.

   - Open the settings menu and select **Tools** > **VISA Conflict Manager**.
   - Under **Enabled Implementations**, make sure that your VISA vendor is selected.
   - Under **Preferred Implementation**, make sure that your VISA vendor is selected.
2. If you are using SCPI commands, check if your device responds to them as expected when they are issued from the configuration utility.
3. Use your VISA vendor's configuration utility to make sure that your device hardware is being detected. You can also check that your device responds to a `*IDN?` query.
4. Use your VISA vendor's debug utility to check the Instrument I/O traffic for errors other than timeout errors.
5. Try installing a different supported VISA vendor's driver.

Interface-Specific VISA

You can use other Instrument Control Toolbox interfaces to troubleshoot VISA. Try connecting to your devices using these interfaces.

- VISA-TCP/IP — `tcpclient` interface
- VISA-Serial — `serialport` interface

See Also
`visadev`

Related Examples

- “Troubleshooting VISA Interface” on page 21-33
- “Troubleshooting GPIB Interface” on page 21-19

Instrument Control Toolbox Examples

- “Recording an Instrument Session” on page 22-3
- “Reading Waveforms from an Oscilloscope Using a Quick-Control Oscilloscope Object” on page 22-9
- “Creating and Downloading an Arbitrary Waveform to a Function Generator” on page 22-12
- “Restoring the BlinkM to Its Factory Settings Using I2C Bus” on page 22-15
- “Communicating with EEPROM using SPI bus” on page 22-18
- “Communicating with the Lego® Mindstorms® NXT brick over Bluetooth®” on page 22-20
- “Tap Detection with ADXL345 Accelerometer Chip Using the NI USB 8451 Adaptor” on page 22-24
- “Reading Inphase and Quadrature (IQ) Data from a Signal Analyzer over TCP/IP” on page 22-29
- “Fetch Waveform through NI-SCOPE MATLAB Instrument Driver in Simulation Mode” on page 22-35
- “Using NI-FGEN Instrument Driver To Generate A Sine Wave” on page 22-39
- “Read Waveform Data from Keysight® DSO-X 2002A Oscilloscopes Using the IVI-C Driver” on page 22-41
- “Read Waveforms from a Keysight® M9210A Digitizer using the IVI-C Driver” on page 22-44
- “Set Output Voltage and Make Measurements on Keysight® AC6801A Power Supply Using the IVI-C Driver” on page 22-48
- “Measure Frequency on Keysight® 532xx Frequency Counter Using the IVI-C Driver” on page 22-54
- “Measure AC Voltage on a Keysight® 34410A Digital Multimeter Using the IVI-C Driver” on page 22-59
- “Set Output Voltage and Make Measurements from a Keysight® AgE3633A DC Power Supply Using the IVI-C Driver” on page 22-62
- “Generate AM Waveforms on Keysight® 3352x Waveform Generator Using the IVI-C Driver” on page 22-65
- “Measure Power on a Keysight® RF Power Meter Using the IVI-C Driver” on page 22-68
- “Configure Output Signal on Keysight® RF Signal Generator Using the IVI-C Driver” on page 22-71
- “Set and Measure DAC (Data Acquisition) Channel Voltage on Keysight® 34970A Switch Using the IVI-C Driver” on page 22-74
- “Acquire Signal Spectrum on a Rohde & Schwarz® Spectrum Analyzer Using the IVI-C Driver” on page 22-77
- “Basic UDP Communication” on page 22-81
- “Video Surveillance Over TCP/IP Network” on page 22-83
- “802.11 OFDM Beacon Frame Generation and Transmission with Test and Measurement Equipment” on page 22-85
- “LTE Waveform Generation and Transmission Using Quick Control RF Signal Generator” on page 22-89

- "Creating and Downloading an IQ Waveform to a RF Signal Generator" on page 22-93
- "Fetch Spectrum Through Ocean Optics Spectrometer Using MATLAB Instrument Driver" on page 22-96
- "Read Streaming Data from Arduino Using Serial Port Communication" on page 22-99
- "Read Waveform from Tektronix TDS 1002 Scope Using SCPI Commands " on page 22-102
- "Read Voltage Through NI-DMM MATLAB Instrument Driver in Simulation Mode" on page 22-105
- "Using a NI Switch Module and a NI DMM to Perform Resistance Measurements" on page 22-108
- "Generate DC Voltage Using NI-DCPOWER MATLAB Instrument Driver in Simulation Mode" on page 22-112
- "Send and Receive Multicast Data Packets Using the User Datagram Protocol" on page 22-115
- "Communicate Between Two MATLAB Sessions Using User Datagram Protocol" on page 22-118
- "Broadcast User Datagram Protocol Data Packets" on page 22-122
- "Communicate Binary and ASCII Data to an Echo Server Using TCP/IP" on page 22-124
- "Communicate Between a TCP/IP Client and Server in MATLAB" on page 22-127
- "Read Data from Arduino Using TCP/IP Communication" on page 22-132
- "Generate a Swept Sinusoid Using VISA and Capture Waveform Using Quick-Control Oscilloscope" on page 22-136
- "5G NR Waveform Acquisition and Analysis" on page 22-140
- "TCP/IP Client Block Communication with Arduino Server" on page 22-152

Recording an Instrument Session

Introduction

This example shows how to record data and event information with an oscilloscope over the serial port interface. However, any interface object can be used with the commands given throughout the example. The instrument used was a Tektronix® TDS 210 oscilloscope.

Functions and Properties

This function is used to record data and event information:

```
RECORD       - Record data and event information to a file.
```

These properties are associated with recording data and event information:

```
RecordDetail - Specifies the amount of information recorded.
RecordMode   - Specifies whether data and event information
               are saved to one record file or to multiple
               record files.
RecordName   - Specifies the name of the record file.
RecordStatus - Indicates if data and event information are
               saved to a record file.
```

Creating a Serial Port Object

To begin, create a serial port object associated with the COM1 port.

```
s=serial('COM1')
```

```
   Serial Port Object : Serial-COM1

   Communication Settings
      Port:               COM1
      BaudRate:           9600
      Terminator:         'LF'

   Communication State
      Status:             closed
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
      BytesAvailable:     0
      ValuesReceived:     0
      ValuesSent:         0
```

Using RECORD

You initiate and terminate recording with the RECORD function. Before recording can begin, the interface object must be connected to the instrument with the FOPEN function. If an error occurs while writing information to the record file, recording will be terminated and a warning will be displayed. When the interface object is closed with the FCLOSE function, recording will automatically be terminated.

The object's RecordStatus property indicates if data and events are being recorded. RecordStatus can be either on or off. The value of the RecordStatus property is configured with the RECORD function.

You can specify the name of the record file with the object's RecordName property. The default value is record.txt.

```
s.RecordStatus
```

```
ans =

off
```

```
fopen(s)
record(s)
s.RecordStatus
```

```
ans =

on
```

```
s.RecordName
```

```
ans =

record.txt
```

Specifying the Amount of Information Recorded

The RecordDetail property specifies the amount of information recorded. RecordDetail can be set to either compact or verbose.

If RecordDetail is set to compact, the following information is captured:

- The number of values read
- The data type of the values read
- The number of values written
- The data type of the values written
- The event information

If RecordDetail is set to verbose, the data read from the instrument and the data written to the instrument are also captured in the record file.

The default value for the RecordDetail property is compact.

Recording ASCII Data

Now query the instrument's identification information. Because recording is on, this information will be captured in the record file.

Note the legend at the top of the record file uses a

- > to indicate data that is written to the instrument
- < to indicate data read from the instrument
- * to indicate events that occurred

```
fprintf(s, '*IDN?')
data = fscanf(s)


data =

TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04



type record.txt


Legend:
  * - An event occurred.
  > - A write operation occurred.
  < - A read operation occurred.

1      Recording on 31-May-2005 at 12:18:38.541. Binary data in little endian format.
2      > 6 ascii values.
3      < 56 ascii values.
```

Now let's capture the data written to the instrument and the data read from the instrument.

```
set(s, 'RecordDetail', 'verbose')
fprintf(s, 'Display:Contrast?')
data = fscanf(s)


data =

50



type record.txt


Legend:
  * - An event occurred.
  > - A write operation occurred.
  < - A read operation occurred.

1      Recording on 31-May-2005 at 12:18:38.541. Binary data in little endian format.
2      > 6 ascii values.
3      < 56 ascii values.
4      > 18 ascii values.
       Display:Contrast?

5      < 3 ascii values.
       50
```

Recording Binary Data

Binary data with uchar, schar, (u)int8, (u)int16, or (u)int32 precision is recorded in the record file in hexadecimal format.

```
fprintf(s, 'Display:Contrast?')
fread(s, 1, 'int16')
```

```
ans =

       12341
```

```
dec2hex(12339)
```

```
ans =

3033
```

```
fclose(s)
type record.txt
```

```
Legend: 
  * - An event occurred.
  > - A write operation occurred.
  < - A read operation occurred.

1      Recording on 31-May-2005 at 12:18:38.541. Binary data in little endian format.
2      > 6 ascii values.
3      < 56 ascii values.
4      > 18 ascii values.
       Display:Contrast?

5      < 3 ascii values.
       50

6      > 18 ascii values.
       Display:Contrast?

7      < 1 int16 values.
       b035
8        Recording off.
```

Binary data with single or double precision is recorded according to the IEEE® 754 floating-point bit layout.

This means that a single precision value is represented as a 32-bit value, which will be converted to the equivalent hex value. To translate the single-precision value, one would have to do the following (bit 1 is the leftmost bit):

```
sign        = bit1 (a value of 0 is positive and a
              value of 1 is negative).
exp         = bit2 to bit 9
```

```
significand = bit 10 to bit 32
value       = (2^(exp-127))*(1.significand)
```

For double-precision values the following would be used (bit 1 is the leftmost bit):

```
sign        = bit1 (a value of 0 is positive and a
                   value of 1 is negative).
exp         = bit2 to bit 12
significand = bit 13 to bit 64
value       = (2^(exp-1023))*(1.significand)
```

Additionally, a text representation of the value will be listed to the right of the single-precision hex value, using the %g format string.

Appending Data to an Existing File

Since recording was terminated, the record file will be overwritten if recording is once again initiated. This is because the default value for RecordMode is overwrite. To avoid overwriting the previous record file, either specify a new value for the RecordName property or set the RecordMode property to append.

```
s.RecordMode = 'append';
fopen(s);
record(s, 'on')
fprintf(s, 'RS232:BAUD?')
data = fscanf(s)
```

```
data =

9600
```

```
fclose(s)
type record.txt
```

```
Legend:
  * - An event occurred.
  > - A write operation occurred.
  < - A read operation occurred.

1      Recording on 31-May-2005 at 12:18:38.541. Binary data in little endian format.
2      > 6 ascii values.
3      < 56 ascii values.
4      > 18 ascii values.
       Display:Contrast?

5      < 3 ascii values.
       50

6      > 18 ascii values.
       Display:Contrast?

7      < 1 int16 values.
       b035
8        Recording off.
```

```
1      Recording on 31-May-2005 at 12:18:41.885. Binary data in little endian format.
2      > 12 ascii values.
       RS232:BAUD?

3      < 5 ascii values.
       9600

4        Recording off.
```

Reading Waveforms from an Oscilloscope Using a Quick-Control Oscilloscope Object

This example shows how to use the Quick-Control Oscilloscope to acquire waveforms from an oscilloscope.

Instrument Control Toolbox™ software supports communication with instruments through Quick-Control Instrument objects. In this example you will learn to acquire a waveform from a Keysight Technologies® (formerly Agilent Technologies®) MSO6014 mixed signal oscilloscope using a Quick-Control oscilloscope object.

For a complete list of supported hardware, visit the Instrument Control Toolbox product page.

Introduction

This example is tested on a 32-bit Microsoft® Windows® system, National Instruments® Compliance Package 4.1 . Keysight I/O Suite and 546XX IVI-C driver version 1.3.20.0, which can be downloaded from Keysight's website: http://www.keysight.com. Ensure that the VISA utility has been set up to recognize the instrument resource before you execute this example.

Creating an Oscilloscope

Before acquiring any data, you must create an oscilloscope instance.

```
myScope = oscilloscope()
```

```
myScope =

oscilloscope: No connection has been setup with instrument, type help oscilloscope for more infor
```

Discovering Available Resources

Find out available resources. A resource is a string identifier to the instrument. You have to set it before connecting to the instrument.

```
availableResources = getResources(myScope)
```

```
availableResources =

 TCPIP0::a-m6104a-004598.dhcp.mathworks.com::inst0::INSTR
```

Connecting to the Oscilloscope Object

If multiple resources are available, use a VISA utility to verify the correct resource and set it.

```
myScope.Resource = 'TCPIP0::a-m6104a-004598::inst0::INSTR';

% Connect to the instrument.
connect(myScope);
```

Examine the Current Oscilloscope Setting

```
get(myScope);
```

```
          AcquisitionTime: 0.0100
    AcquisitionStartDelay: -0.0050
             TriggerLevel: 0.1000
             TriggerSlope: 'rising'
            TriggerSource: 'Channel1'
           WaveformLength: 2000
              TriggerMode: 'normal'
          SingleSweepMode: 'on'
             ChannelNames: {'Channel1'  'Channel2'  'Channel3'  'Channel4'}
          ChannelsEnabled: {'Channel1'}
                 Resource: 'TCPIP0::a-m6104a-004598::inst0::INSTR'
                   Driver: 'Ag546XX'
      DriverDetectionMode: 'auto'
                  Timeout: 10
                   Status: 'open'
```

Configuring the Oscilloscope

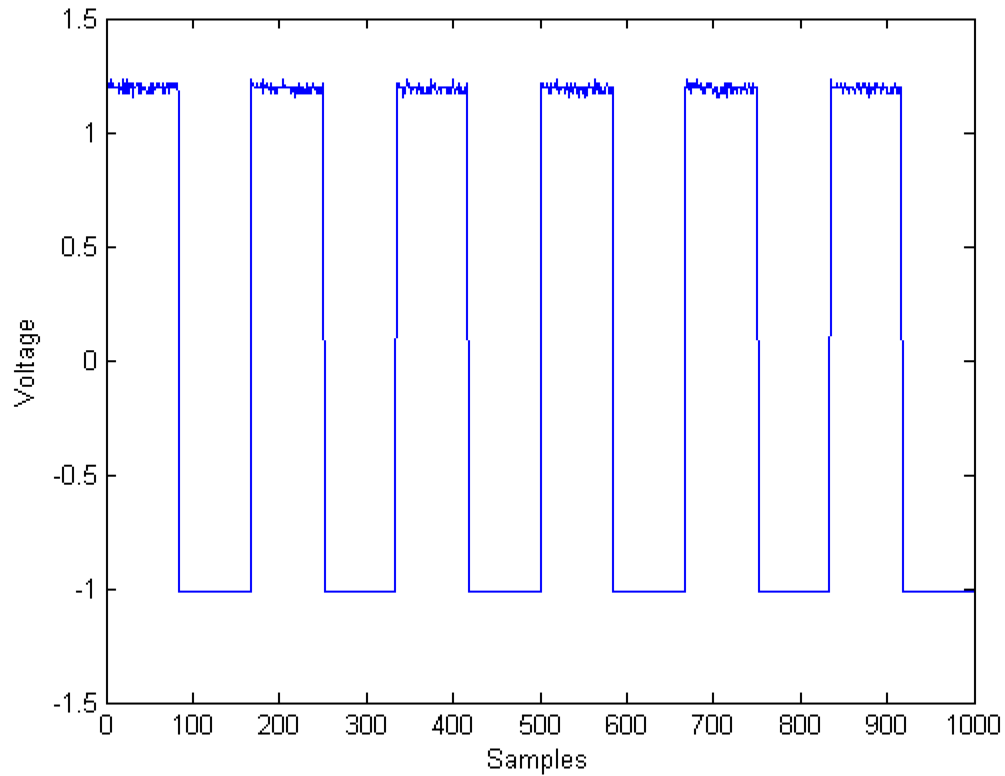
Configure the oscilloscope's settings. The configuration we will use in this example is: acquisition time of 0.01 second with 2000 data points, trigger level of 0.1v and normal trigger mode, channel one enabled and vertical settings as shown below.

```
% Automatically configuring the instrument based on the input signal.
autoSetup(myScope);

myScope.AcquisitionTime = 0.01;

myScope.WaveformLength = 2000;

myScope.TriggerMode = 'normal';

myScope.TriggerLevel = 0.1;

enableChannel(myScope, 'Channel1');

setVerticalCoupling (myScope, 'Channel1', 'AC');

setVerticalRange (myScope, 'Channel1', 5.0);
```

Getting a Waveform from Channel One

This function initiates an acquisition on the enabled channel. It then waits for the acquisition to complete and returns the waveform for the specified channel.

```
waveformArray = getWaveform(myScope, 'acquisition', true);

% Plot the waveform.
plot(waveformArray);
xlabel('Samples');
ylabel('Voltage');
```



Cleaning Up

Once you have finished configuring the instrument and retrieved data from it, you need to close the connection and remove it from the workspace.

```
disconnect(myScope);
clear myScope;
```

Creating and Downloading an Arbitrary Waveform to a Function Generator

This example shows how to use the Quick-Control Function Generator to generate arbitrary waveforms.

Instrument Control Toolbox™ supports communication with instruments through interfaces and drivers.

For a complete list of supported hardware, visit the Instrument Control Toolbox product page.

Introduction

In this example we will create and download an arbitrary waveform to an arbitrary waveform generator using Quick-Control Function Generator.

Requirements

To run this example you need:

- An arbitrary waveform generator (This example uses Tektronix® AFG3022B).
- VISA software installed on your machine (This example uses Agilent® IO Libraries Version 16.1).
- IVI-C drivers for the instruments installed on your machine (This example uses Tektronix® tkafg3k IVI-C driver version 3.2 from the IVI Foundation Driver Registry).
- Instrument Control Toolbox™.

Define waveform parameters

We will create an arbitrary waveform that consists of three different waveforms. Each waveform's properties, including the amplitude and frequency,will be set in the section below. For each waveform, the amplitude is in `volts`, while the frequency is in `Hz`.

When generating signals for a function generator it is important to ensure continuity in the time domain so as to not introduce unintended spectral content in the signal, especially if the waveform is going to be played back repeatedly. To ensure continuity you can define the time vector such that it contains an integral number of cycles of each of the three tones that will compose the synthesized waveform.

```
timeStep = 0.001;
time = 0:timeStep:(1-timeStep);
```

Parameters for the first waveform

```
amplitude1 = 0.2;
frequency1 = 10;
```

Parameters for the second waveform

```
amplitude2 = 0.8;
frequency2 = 14;
```

Parameters for the third waveform

```
amplitude3 = 0.6;
frequency3 = 18;
```

Create arbitrary waveform

We will create our three individual waveforms using the `sin` command.

```
waveform1 = amplitude1*sin(2*pi*frequency1*time);
waveform2 = amplitude2*sin(2*pi*frequency2*time);
waveform3 = amplitude3*sin(2*pi*frequency3*time);
```

The arbitrary waveform will be a combination of each of the above listed waveforms.

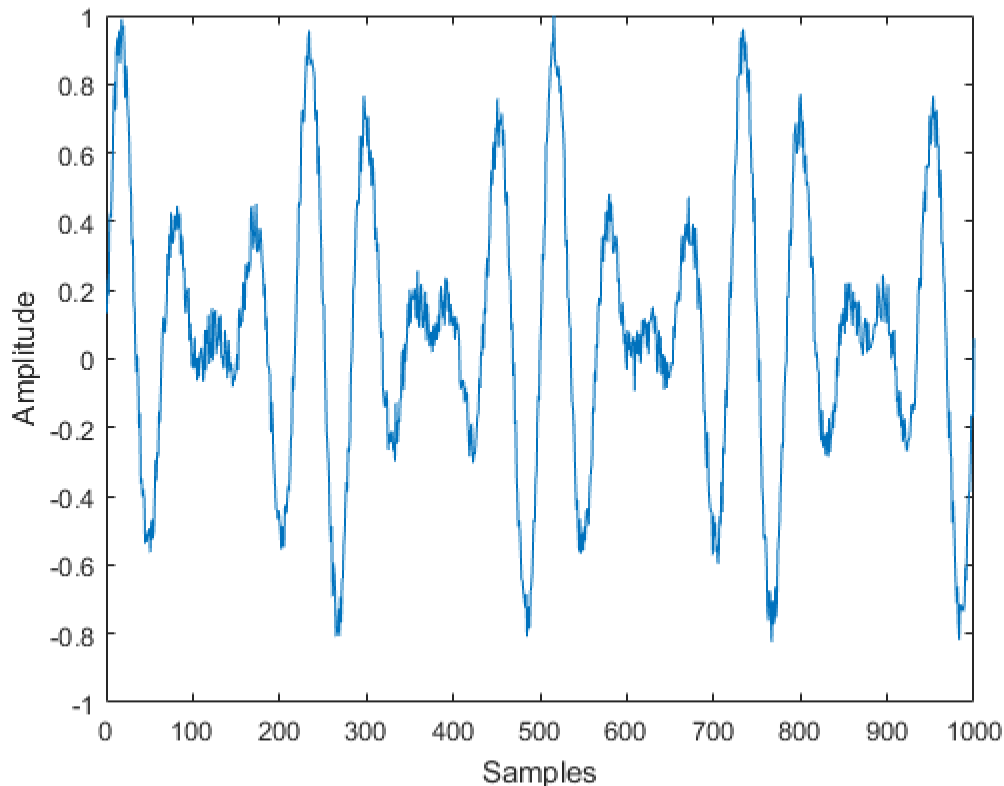
```
waveform = waveform1 + waveform2 + waveform3;
```

Add random noise to the waveform created earlier.

```
waveform = waveform + 0.3*rand(1,size(waveform,2));
```

Some function generators require a normalized waveform. In this case the waveform is normalized between -1 to +1.

```
waveformArray = (waveform./max(waveform))';
plot(waveformArray);
xlabel('Samples');
ylabel('Amplitude');
```



Discover all the available instrument resources (targets) you can connect to, using the `resources` command.

```
f = fgen;
f.resources
```

```
ans =

  1×211 char array

 ASRL1::INSTR
 ASRL3::INSTR
 ASRL::COM1
 ASRL::COM3
 GPIB0::INTFC
 PXI0::MEMACC
 TCPIP0::172.28.22.217::inst0::INSTR
 TCPIP0::172.28.23.55::inst0::INSTR
 TCPIP0::a-d6054l-000006.dhcp.mathworks.com::inst0::INSTR
```

Connect to function generator

Now that you have your waveform you need to download it onto the function generator. You will use the Quick-Control Function Generator or `fgen` function of the Instrument Control Toolbox™ to do so. Because the IP address of the instrument is *172.28.22.217*, the resource specified will be TCPIP0::172.28.22.217::inst0::INSTR.

```
f = fgen('TCPIP0::172.28.22.217::inst0::INSTR','tkafg3k');
```

Download the created waveform

Specify the channel of the function generator where the waveform would be generated. Here, the waveform would be generated on channel number 1.

```
selectChannel(f, '1');
```

Since you will be generating a custom arbitrary waveform, set the `Waveform` property to *Arb*.

```
f.Waveform = 'Arb';
```

You are now ready to download the previously generated arbitrary waveform *waveformArray* to the function generator.

```
downloadWaveform (f, waveformArray);
```

Enable the waveform generation.

```
enableOutput(f);
```

Clean up

```
clear f;
```

Restoring the BlinkM to Its Factory Settings Using I2C Bus

This example shows how to restore the BlinkM RGB LED to its factory settings by communicating with it over the I2C bus.

To begin, create an I2C object. For this example, we are using the BlinkM RGB LED which has board index as 0 and address (hex) as 9h. To connect the computer to the I2C bus, a USB-I2C adaptor from Aardvark is used. For I2C object creation, this translates to:

- `BoardIndex` = 0
- `RemoteAddress` = 9h
- `Vendor` = Aardvark.

```
blinkM = i2c('aardvark',0,hex2dec('09'))
```

```
   I2C Object : I2C-0-9h

   Communication Settings
      BoardIndex          0
      BoardSerial         2237481561
      BitRate:            100
      RemoteAddress:      9h
      Vendor:             aardvark


   Communication State
      Status:             closed
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
```

Before you can perform a read or write operation, you must connect the I2C object to the device with the `fopen` function. If the object was successfully connected, its Status property is automatically configured to open.

```
fopen(blinkM);
get(blinkM, 'Status')
```

```
ans =

open
```

The default factory setting for the BlinkM is to play the colors white-red-green-blue-off in a sequence. The white color stays for 100 clock ticks while the colors red, green, blue and the off state stay for 50 clock ticks. This example uses a script to play this sequence on the BlinkM RGB LED.

The BlinkM data sheet specifies that the script length can be set by writing 'L' to the device followed by 3 arguments. We will be using the `fwrite` function to write binary data onto the I2C bus.

- First argument - As we want the default factory settings, use the script with ID 0.

- Second argument - Length of the script, this script has 6 lines of code.
- Third argument - Number of the repeats of the script, to play the script forever, we set the number of repeats to be 0.

```
fwrite(blinkM,['L' 0 6 0]);
```

Note: The eeprom write time is approximately 20 ms for the BlinkM. Hence you might have to pause to ensure that the command is written successfully.

According to the data sheet, a script line can be written to the device by writing 'W' followed by 7 arguments.

- First argument - Script ID, 0 in this case.
- Second argument - Line number you want to write.
- Third argument - Duration in ticks that the command will last.
- Arguments 4:7 - Actual BlinkM command and its arguments.

To start with, first set the fade speed for the device by writing 'f' followed by the speed to the device.

```
fwrite(blinkM,['W' 0 0 1 'f' 10 0 0]);
```

In the next 5 commands, specify the color sequence to be white, red, green, blue and off. A color can be set on the LED by writing 'n' to it followed by the R, G and B values. E.g., set the first color in the sequence to be white by setting the R, G, B values to 255.

```
fwrite(blinkM,['W' 0 1 100 'n' 255 255 255]);

fwrite(blinkM,['W' 0 2 50 'n' 255 0 0]);

fwrite(blinkM,['W' 0 3 50 'n' 0 255 0]);

fwrite(blinkM,['W' 0 4 50 'n' 0 0 255]);

fwrite(blinkM,['W' 0 5 50 'n' 0 0 0]);
```

The BlinkM data sheet specifies that the command to start up (or 'boot') the device is 'B' followed by 5 arguments.

- First argument - Set to 1 to play a script.
- Second argument - Script ID, 0 in this case.
- Third argument - Number of the repeats of the script, to play the script forever, we set the number of repeats to be 0.
- Fourth argument - Fade speed.
- Fifth argument - Time adjust.

```
fwrite(blinkM,['B' 1 0 0 8 0]);
```

The previously written script can be played by writing 'p' followed by 3 arguments.

- First argument - Script ID, 0 in this case.
- Second argument - Number of the repeats of the script, to play the script forever, we set the number of repeats to be 0.

- Third argument - Script line number to start playing from.

  ```
  fwrite(blinkM,['p' 0 0 0]);
  ```

Disconnect it from the device, remove it from memory, and remove it from the workspace.

```
fclose(blinkM);
delete(blinkM);
clear ('blinkM');
```

This concludes the example for restoring the BlinkM to its default state by writing a script to it. This script can be updated to play customised sequences on the BlinkM RGB LED.

Communicating with EEPROM using SPI bus

This example shows how to communicate with EEPROM AT25080A on Aardvark's I2C/SPI Activity Board over the Serial Peripheral Interface (SPI) bus.

To begin, create an SPI object. For this example, we are using Aardvark's I2C/SPI Activity Board which has both board index and address as 0. To connect the computer to the SPI bus, a USB-I2C/SPI adaptor from Aardvark is used. For SPI object creation, this translates to:

- `Vendor` = aardvark
- `BoardIndex` = 0
- `Port` = 0

```
eeprom = spi('aardvark',0,0);
disp(eeprom);
```

```
SPI Object :

    Adapter Settings
        BoardIndex:             0
        BoardSerial:            2237727782
        VendorName:             aardvark

    Communication Settings
        BitRate:                1000000 Hz
        ChipSelect:             0
        ClockPhase:             FirstEdge
        ClockPolarity:          IdleLow
        Port:                   0

    Communication State
        ConnectionStatus:       Disconnected

    Read/Write State
        TransferStatus:         Idle
```

Before you can perform a read or write operation, you must connect the SPI object to the device using the `connect` function. You can verify that the object has been successfully connected, by checking its `ConnectionStatus` property. Once connected to the device, the property `ConnectionStatus` is automatically updated to be `Connected`.

```
connect(eeprom);
eeprom.ConnectionStatus
```

```
ans =

Connected
```

SPI operates in full duplex mode. Hence for any read/write operation, data is always transferred in both directions. This can be illustrated with a simple task of writing 'Hello' to EEPROM and reading it back.

The EEPROM's datasheet specifies the following for reading and writing data:

- The chip should be write-enabled before writing anything to it.The chip can be write-enabled by writing `6` to it. NOTE: If the chip is not write-enabled, it will ignore the write instruction and will return to the standby state
- Data should be written to the chip in the following format:

```
[Write_Command Upper_Byte_Address Lower_Byte_Address data1 data2 ...]
```

The Write_Command for this EEPROM is 2.

- Data should be written to the chip in the following format to read it back correctly:

```
[Read_Command Upper_Byte_Address Lower_Byte_Address zeros(1,size of data to be read back)]
```

The Read_Command for this EEPROM is 3.

Write-enable the eeprom

```
write(eeprom,6);
```

Write 'Hello' at the 0th address of EEPROM using the `write` function.

```
dataToWrite = [2 0 0 double('Hello')];
write(eeprom, dataToWrite);
```

We can now read data back from EEPROM.

```
dataToWrite = [3 0 0 zeros(1,5)];
returnedData = writeAndRead(eeprom, dataToWrite);
```

The data returned is:

- Bytes 1:3 - Don't care
- Fourth byte onwards - Data read back from EEPROM

In this case, the data read back is:

```
char(returnedData(4:end))
```

```
ans =

Hello
```

Disconnect the SPI object and remove it from memory and the workspace.

```
disconnect(eeprom);
clear('eeprom');
```

Communicating with the Lego® Mindstorms® NXT brick over Bluetooth®

This example illustrates communication with a Lego Mindstorms NXT brick using text commands sent over the Bluetooth Serial Port Profile.

Get information about available Bluetooth devices

Before setting up a connection in MATLAB, the NXT brick has to be paired with your computer. Once this is done, the device will be visible in MATLAB.

To get a list of all the available Bluetooth devices use the `instrhwinfo` command. Get the "friendly" names of the Bluetooth devices available using the `RemoteNames` field from the output of `instrhwinfo`.

```
bluetoothDevices = instrhwinfo('Bluetooth')

remoteNames = bluetoothDevices.RemoteNames
```

```
bluetoothDevices = 

        RemoteNames: {'NXT'}
          RemoteIDs: {'btspp://00165310E7C4'}
    BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
     JarFileVersion: 'Version 3.2'


remoteNames = 

    'NXT'
```

View details of the NXT brick

Get the details of the NXT brick.

```
deviceInfo = instrhwinfo('Bluetooth','NXT')
```

```
deviceInfo = 

               RemoteName: 'NXT'
                 RemoteID: 'btspp://00165310E7C4'
    ObjectConstructorName: {'Bluetooth('NXT', 1);'}
                 Channels: {'1'}
```

Set up a connection to the NXT brick

The device's `RemoteName` is `NXT` and its `Channel` is `1`. Construct a bluetooth object called `bt` as shown by the `ObjectConstructorName` in the previous step.

```
bt = Bluetooth('NXT', 1);
```

Now connect to the device.

```
fopen(bt);
```

Get information about the Lego Mindstorms NXT brick

The Lego Mindstorms data sheet defines a protocol for communicating with NXT brick. It specifies that to get the device information, the command should have following format:

- Bytes 1:2 - Length of command
- Byte 3 - Type of command
- Byte 4 - Additional information to execute the command

Using this information, raw data packet is formed which is sent to the NXT brick.

- Bytes 1:2 - Length of command = 2
- Byte 3 - Type of command = 1 (since it is a system command with reply from device)
- Byte 4 - 0x9B (This value is defined by the Lego Mindstorms NXT communication protocol)

This translates the raw data packet to be: `[2 0 1 155]`. Write this data to the brick.

```
fwrite(bt,[2 0 1 155]);
```

The device then gives a response containing device information. The response has following format:

- Bytes 1:2 - Length of response
- Byte 3 - Type of command issued
- Byte 4 - Data sent to device
- Byte 5 - Command status
- Bytes 6:20 - Name of device (14 characters + null terminator)
- Bytes 21:27 - Bluetooth address
- Byte 28 - LSB of Bluetooth signal strength
- Byte 29 - Not defined
- Byte 30 - Not defined
- Byte 31 - MSB of Bluetooth signal strength
- Byte 32 - LSB of user flash
- Byte 33 - Not defined
- Byte 34 - Not defined
- Byte 35 - MSB of user flash

Now read the response from the brick to get its information

```
btResponse = fread(bt,32)'
```

```
btResponse =

  Columns 1 through 13

    33     0     2   155     0    78    88    84     0     0     0     0     0

  Columns 14 through 26
```

```
     0     0     0     0     0     0     0     0    22    83    16   231   196

  Columns 27 through 32

     0     0     0     0     0   132
```

In this example, the response from the NXT brick is:

- Bytes 1:2 - Length of response = 33
- Byte 3 - Type of command issued = 2
- Byte 4 - Data sent to device = 0x9B
- Byte 5 - Command status = 0 (command was successful)
- Bytes 6:20 - Name of device (14 characters + null terminator)
- Bytes 21:27 - Bluetooth address
- Byte 28 - LSB of Bluetooth signal strength = 0
- Byte 29 - Not defined = 0
- Byte 30 - Not defined = 0
- Byte 31 - MSB of Bluetooth signal strength = 0
- Byte 32 - LSB of user flash = 0
- Byte 33 - Not defined = 0
- Byte 34 - Not defined = 0
- Byte 35 - MSB of user flash = 132

Let us now read only the name of the brick from its response. We know that bytes 6 to 20 correspond to the brick's name.

```
btName = char(btResponse(6:20))
```

```
btName =

NXT
```

Clear the buffer of the Bluetooth object

```
flushinput(bt);
```

Play a tone on Lego Mindstorms NXT brick

Let us now play a tone at frequency 1500 Hz for 1000 ms on the brick. The Lego Mindstorms NXT direct commands define a command to play a tone on the brick as follows:

- Bytes 1:2 - Length of command
- Byte 3 - Type of command
- Byte 4 - Message id
- Bytes 5:6 - Frequency of the tone in Hz
- Bytes 7:8 - Duration of the tone in ms

In this example, the raw data packet is formed as follows:

- Byte 1:2 - Length of command = 6
- Byte 3 - Type of command = 0 (since it is a direct command with reply from device)
- Byte 4 - Message id = 3
- Bytes 5:6 - Frequency of the tone is 1500Hz, which translates to uint8[220 5]
- Bytes 7:8 - Duration of the tone is 1000 ms, which translates to uint8[232 3]

Hence the command to be written to the brick is `[6 0 0 3 220 5 232 3]`.

```
fwrite(bt,[6 0 0 3 220 5 232 3]);
```

In addition to the audio tone played on the NXT brick, the device also gives a 5 byte response. Let us now read the response from the brick. The response has the following format:

- Byte 1:2 - Length of response
- Byte 3 - Type of command issued
- Byte 4 - Message id
- Byte 5 - Status byte

```
btResponse = fread(bt,5)'
```

```
btResponse =

     3     0     2     3     0
```

In this example, the response from the NXT brick is:

- Byte 1:2 - Length of response = 3
- Byte 3 - Type of command issued = 2
- Byte 4 - Message id = 3
- Byte 5 - Status byte = 0 (command was successful)

Close the connection to the NXT brick

```
fclose(bt);
delete(bt);
clear('bt');
```

You can get more information on communicating with Lego Mindstorms NXT using high level commands here.

Tap Detection with ADXL345 Accelerometer Chip Using the NI USB 8451 Adaptor

This example shows how to write and read data from the ADXL345 I2C enabled accelerometer chip using the NI® USB 8451 I2C adaptor. The accelerometer will be configured to detect a double tap and MATLAB® will be used to display a message that the chip has detected this.

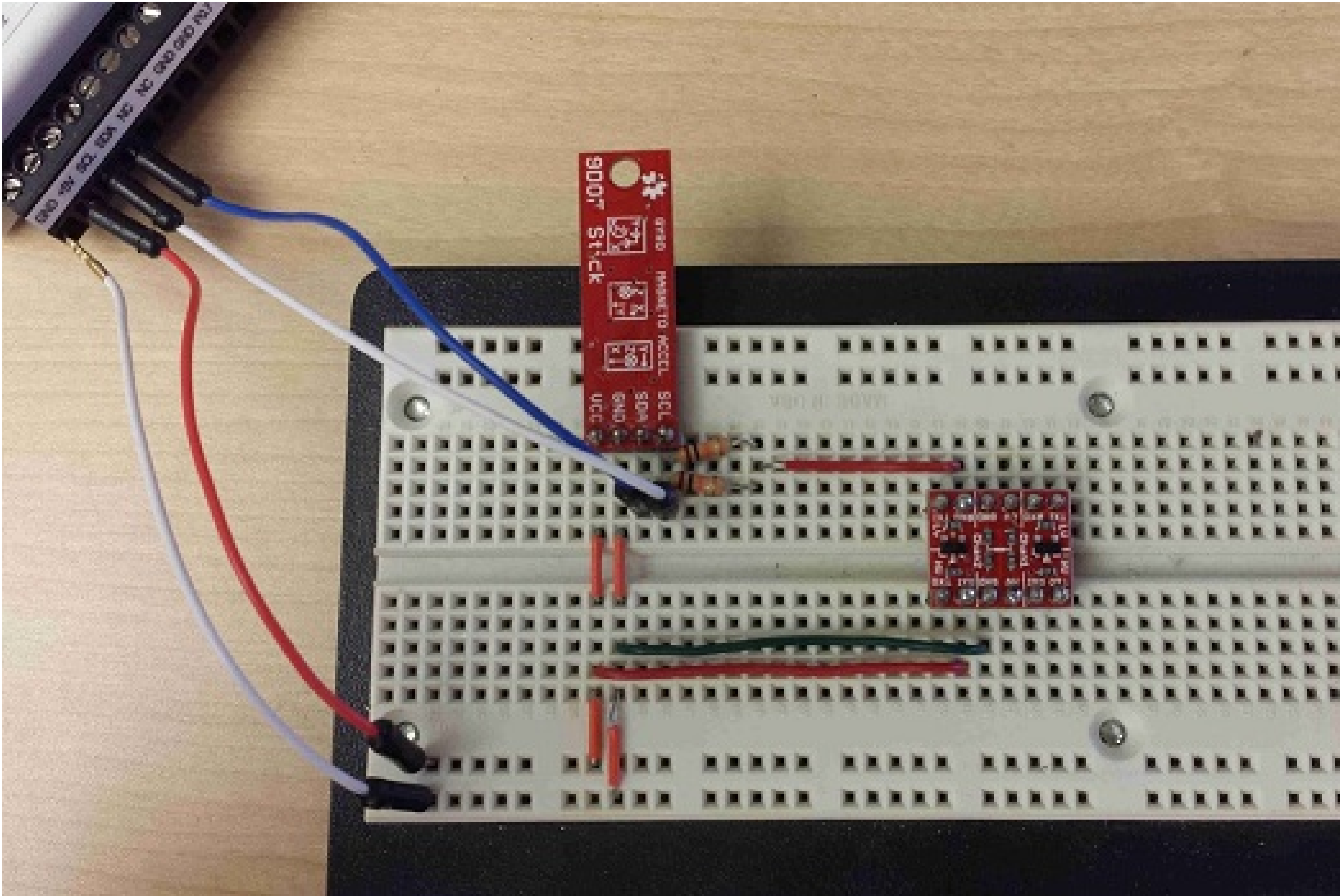
Instrument Control Toolbox™ supports communication with instruments through interfaces and drivers.

For a complete list of supported hardware, visit the Instrument Control Toolbox product page.

Requirements

This example requires a Microsoft® Windows® system and NI845x driver 2.1.1 or higher installed. Make sure the Measurement & Automation Explorer recognizes the NI845x device before you use this example. In addition, ensure the Measurement & Automation Explorer™ is closed prior to accessing the device form MATLAB.

The example uses the Analog Devices® ADXL345 accelerometer that is mounted on a 9DOF Sensor Stick from SparkFun™. The accelerometer has I2C bus lines that can be connected to the adaptors I2C bus line inputs. The bus lines needs to be pulled up high with external pull up resistors as the NI USB 8451 does not have programmable internal pullup resistors. Note that a level shifter is used to separate V_s and V_dd.



Introduction

Instrument Control Toolbox™ supports communication with I2C devices via I2C adaptors such as the NI USB 8451. The toolbox allows you to create an I2C interface that can be used to configure the adaptor to communicate with the I2C peripheral chips.

This example will demonstrate how to configure an I2C based accelerometer to respond to tapping the breadboard twice with your finger. When the double tap is detected, a message will be displayed in the MATLAB command window.

Verify NI845x Installation

Use the `instrhwinfo` command to check if the NI845x driver is installed correctly and that Instrument Control Toolbox can detect it correctly.

The board serial number should help you identify your device

```
instrreset
i2cInfo = instrhwinfo ('i2c', 'ni845x');
disp(i2cInfo);
```

```
          AdaptorDllName: [1x94 char]
       AdaptorDllVersion: 'Version 3.4'
             AdaptorName: 'ni845x'
```

```
            BoardIdsInUse: [1x0 double]
        InstalledBoardIDs: 0
     DetectedBoardSerials: {'0180D47A (BoardIndex: 0)'}
    ObjectConstructorName: 'i2c('ni845x', BoardIndex, RemoteAddress);'
            VendorDllName: 'Ni845x.dll'
  VendorDriverDescription: 'National Instruments NI USB 845x Driver'
```

Create the I2C Interface and set the bus speed (BitRate property)

```
accelerometerAddress = hex2dec('53');
i2cInterface = i2c('ni845x', 0, accelerometerAddress);
fopen(i2cInterface);
i2cInterface.BitRate = 100;
```

Set the tap threshold, second tap latency, second tap window and tap duration register values

Set the register values according to the datasheet of the device. Read the value back from the device to confirm that the value is indeed set.

```
threshTapRegisterAddress = hex2dec('1D');
valueToWrite = hex2dec('50'); %5g value
disp(['Writing Value: ' num2str(valueToWrite)]);
fwrite(i2cInterface, [threshTapRegisterAddress valueToWrite]);

fwrite(i2cInterface, threshTapRegisterAddress);
registerValue = fread(i2cInterface, 1, 'uint8');
disp(['The value of the THRESH_TAP register is :' num2str(registerValue)]);

latentRegisterAddress = hex2dec('22');
valueToWrite = hex2dec('5');
disp(['Writing Value: ' num2str(valueToWrite)]);
fwrite(i2cInterface, [latentRegisterAddress valueToWrite]);

% Confirm the value return to the register
fwrite(i2cInterface, latentRegisterAddress);
registerValue = fread(i2cInterface, 1, 'uint8');
disp(['The value of the LATENT register is :' num2str(registerValue)]);

windowRegisterAddress = hex2dec('23');
valueToWrite = hex2dec('FF');
disp(['Writing Value: ' num2str(valueToWrite)]);
fwrite(i2cInterface, [windowRegisterAddress valueToWrite]);

% Confirm the value return to the register
fwrite(i2cInterface, windowRegisterAddress);
registerValue = fread(i2cInterface, 1, 'uint8');
disp(['The value of the WINDOW register is :' num2str(registerValue)]);

durationRegisterAddress = hex2dec('21');
valueToWrite = hex2dec('10');
disp(['Writing Value: ' num2str(valueToWrite)]);
fwrite(i2cInterface, [durationRegisterAddress valueToWrite]);

fwrite(i2cInterface, durationRegisterAddress);
registerValue = fread(i2cInterface, 1, 'uint8');
disp(['The value of the DUR register is :' num2str(registerValue)]);
```

```
tapAxesRegisterAddress = hex2dec('2A');
valueToWrite = bin2dec('00000111');
disp(['Writing Value: ' num2str(valueToWrite)]);
fwrite(i2cInterface, [tapAxesRegisterAddress valueToWrite]);

fwrite(i2cInterface, tapAxesRegisterAddress);
registerValue = fread(i2cInterface, 1, 'uint8');
disp(['The value of the TAP_AXES register is :' num2str(registerValue)]);

interruptEnableRegisterAddress = hex2dec('2E');
valueToWrite = bin2dec('01100000');
disp(['Writing Value: ' num2str(valueToWrite)]);
fwrite(i2cInterface, [interruptEnableRegisterAddress valueToWrite]);

fwrite(i2cInterface, interruptEnableRegisterAddress);
registerValue = fread(i2cInterface, 1, 'uint8');
disp(['The value of the INT_ENABLE register is :' num2str(registerValue)]);
```

```
Writing Value: 80
The value of the THRESH_TAP register is :80
Writing Value: 5
The value of the LATENT register is :5
Writing Value: 255
The value of the WINDOW register is :255
Writing Value: 16
The value of the DUR register is :16
Writing Value: 7
The value of the TAP_AXES register is :7
Writing Value: 96
The value of the INT_ENABLE register is :96
```

Enable operation by writing to POWER_CTL register

Writing to the POWER_CTL register as per the datasheet will cause the chip to go from standby mode to normal operation mode.

```
powerControlRegisterAddress = hex2dec('2D');
valueToWrite = bin2dec('00001000');
disp(['Writing Value: ' num2str(valueToWrite)]);
fwrite(i2cInterface, [powerControlRegisterAddress valueToWrite]);

fwrite(i2cInterface, powerControlRegisterAddress);
registerValue = fread(i2cInterface, 1, 'uint8');
disp(['The value of the POWER_CTL register is :' num2str(registerValue)]);
```

```
Writing Value: 8
The value of the POWER_CTL register is :8
```

Poll the interrupt register

The interrupt source register will contain bits that correspond to interrupt flags being generated by specific sources. Check to see that the double tap interrupt is generated

```
interruptSourceRegisterAddress = hex2dec('30');

disp('Waiting for double tap...');
while(1)
```

```
    fwrite(i2cInterface, interruptSourceRegisterAddress);
    InterruptValues = fread(i2cInterface, 1);
    TapInterrupt = bitand(InterruptValues, bin2dec('00100000'));
    if TapInterrupt
        disp('Double tap detected!');
        break;
    end
end
```

```
Waiting for double tap...
Double tap detected!
```

Reading Inphase and Quadrature (IQ) Data from a Signal Analyzer over TCP/IP

This example shows how to acquire IQ Data from a signal analyzer over a TCP/IP interface.

Instrument Control Toolbox™ supports communication with instruments through interfaces and drivers.

For a complete list of supported hardware, visit the Instrument Control Toolbox product page.

Introduction

This example acquires IQ Data from a Keysight Technologies® (formerly Agilent Technologies®) X-Series Signal Analyzer (N9030A, PXA Signal Analyzer) over a TCP/IP interface.

Requirements

To run this example you need an X-Series Signal analyzer with an Ethernet (TCP/IP) connection. You can also execute this example with MATLAB on your X-Series analyzer, or on a PC on the same network as the X-Series Analyzer.

This example uses functions from the Instrument Control Toolbox and the DSP System Toolbox™.

Define Measurement Parameters

Define the parameters used to configure the instrument before you make the measurement. Based on the signal you are measuring, you may need to modify some of the following parameters.

```
% Specify the IP address of the signal analyzer
addressMXA = "172.28.16.61";
```

Parameter Definitions

```
% Center frequency of the modulated waveform (Hz)
centerFrequency = 2.14e9;

% Bandwidth of the signal (Hz)
bandwidth = 25e6;

% Measurement time (s)
measureTime = 8e-3;

% Mechanical attenuation in the signal analyzer(dB)
mechAttenuation = 0;

% Start frequency for Spectrum Analyzer view
startFrequency = 2.11e9;

% Stop frequency for Spectrum Analyzer view
stopFrequency = 2.17e9;

% Resolution Bandwidth for Spectrum Analyzer view
resolutionBandwidth = 200e3;

% Video Bandwidth for Spectrum Analyzer view
videoBandwidth = 300;
```

Connect to the Instrument

- Set up instrument connectivity using a TCP/IP connection.
- Set the timeout to allow sufficient time for the measurement and transfer of data.
- Set the byte order to be "big-endian" to read the floating point data in the correct format from the analyzer.

```
signalAnalyzerObject = tcpclient(addressMXA, 5025);
signalAnalyzerObject.ByteOrder = "big-endian";
signalAnalyzerObject.Timeout = 20;
```

Query Instrument Identification Information

Reset the instrument to a known state using the appropriate SCPI command. Query the instrument identity to make sure we are connected to the right instrument.

```
writeline(signalAnalyzerObject, "*RST");
instrumentInfo = writeread(signalAnalyzerObject, "*IDN?");
disp("Instrument identification information: " + instrumentInfo);
```

```
Instrument identification information: Agilent Technologies,N9030A,US00071181,A.14.16
```

Set Up Instrument for an IQ Waveform Measurement

The X-Series signal and spectrum analyzers perform IQ measurements as well as spectrum measurements. In this example, you acquire the time domain IQ data, visualize it in MATLAB, and perform signal analysis on the acquired data. Use SCPI commands to configure the instrument to make the measurement and define the format of the data transfer once the measurement is made.

```
% Set up signal analyzer mode to Basic/IQ mode
writeline(signalAnalyzerObject,":INSTrument:SELect BASIC");

% Set the center frequency
writeline(signalAnalyzerObject,":SENSe:FREQuency:CENTer " + num2str(centerFrequency));

% Set the resolution bandwidth
writeline(signalAnalyzerObject,":SENSe:WAVEform:BANDwidth:RESolution " + num2str(bandwidth));

% Turn off averaging
writeline(signalAnalyzerObject,":SENSe:WAVeform:AVER OFF");

% Set to take one single measurement once the trigger line goes high
writeline(signalAnalyzerObject,":INIT:CONT OFF");

% Set the trigger to external source 1 with positive slope triggering
writeline(signalAnalyzerObject,":TRIGger:WAVeform:SOURce IMMediate");
writeline(signalAnalyzerObject,":TRIGger:LINE:SLOPe POSitive");

% Set the time for which measurement needs to be made
writeline(signalAnalyzerObject,":WAVeform:SWE:TIME " + num2str(measureTime));

% Turn off electrical attenuation.
writeline(signalAnalyzerObject,":SENSe:POWer:RF:EATTenuation:STATe OFF");

% Set mechanical attenuation level
writeline(signalAnalyzerObject,":SENSe:POWer:RF:ATTenuation " + num2str(mechAttenuation));
```

```
% Turn IQ signal ranging to auto
writeline(signalAnalyzerObject,":SENSe:VOLTage:IQ:RANGe:AUTO ON");

% Set the endianness of returned data
writeline(signalAnalyzerObject,":FORMat:BORDer NORMal");

% Set the format of the returned data
writeline(signalAnalyzerObject,":FORMat:DATA REAL,32");
```

Initiate Measurement

Trigger the instrument to make the measurement, wait for the measurement operation to be complete and read in the waveform. Before you process the data, separate the I & the Q components from the interleaved data returned by the instrument and create a complex vector in MATLAB.

```
% Trigger the instrument and initiate measurement
writeline(signalAnalyzerObject,"*TRG");
writeline(signalAnalyzerObject,":INITiate:WAVeform");

% Wait till measure operation is complete
measureComplete = writeread(signalAnalyzerObject,"*OPC?");

% Read the IQ data
writeline(signalAnalyzerObject,":READ:WAV0?");
data = readbinblock(signalAnalyzerObject,"single");

% Read the additional terminator character from the instrument
read(signalAnalyzerObject,1);

% Separate the data and build the complex IQ vector.
inphase = data(1:2:end);
quadrature = data(2:2:end);
IQData = inphase+1i*quadrature;
```

Display Information About the Measurement

The instrument provides information about the most recently acquired data. Capture this information and display it.

```
writeline(signalAnalyzerObject,":FETCH:WAV1?");
signalSpec = readbinblock(signalAnalyzerObject,"single");
sampleRate = 1/signalSpec(1);
disp("Sample Rate (Hz) = " + num2str(sampleRate));
disp("Number of points read = " + num2str(signalSpec(4)));
disp("Max value of signal (dBm) = " + num2str(signalSpec(6)));
disp("Min value of signal (dBm) = " + num2str(signalSpec(7)));
```

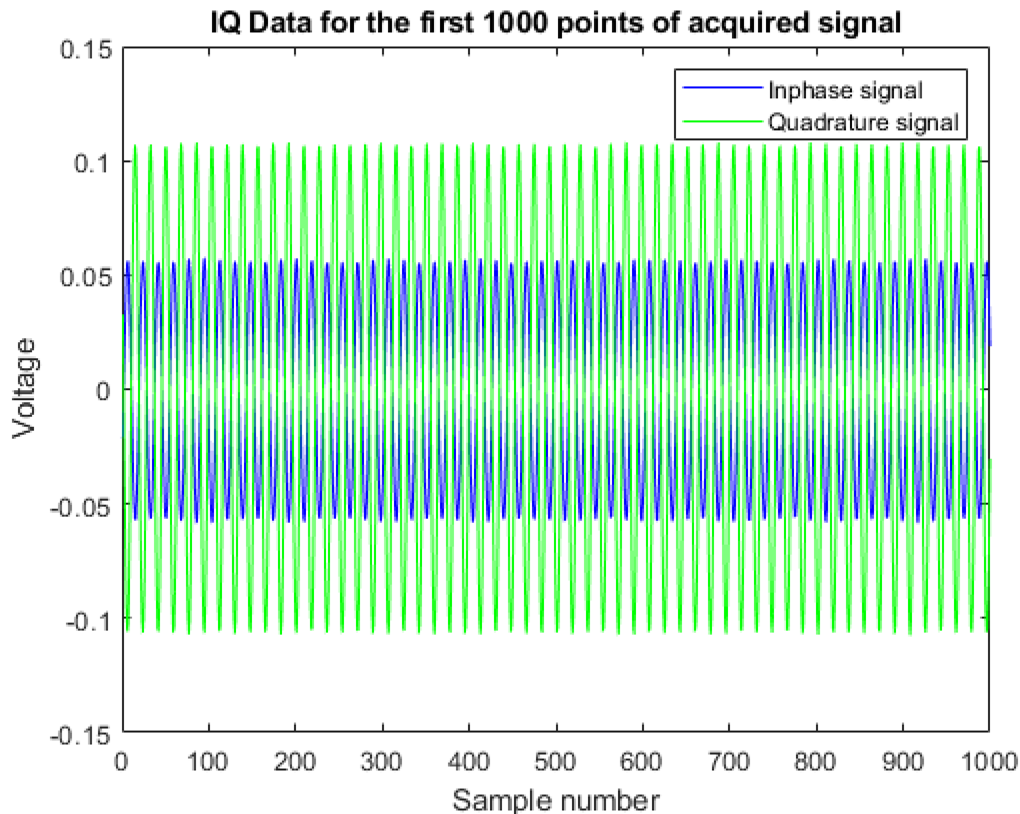
```
Sample Rate (Hz) = 31250000.8838
Number of points read = 250001
Max value of signal (dBm) = -8.211
Min value of signal (dBm) = -42.5689
```

Plot the Acquired IQ Data

Plot the first 1000 points of acquired time domain data and annotate the figure.

```
figure(1)
plot(real(IQData(1:1000)),"b");
```

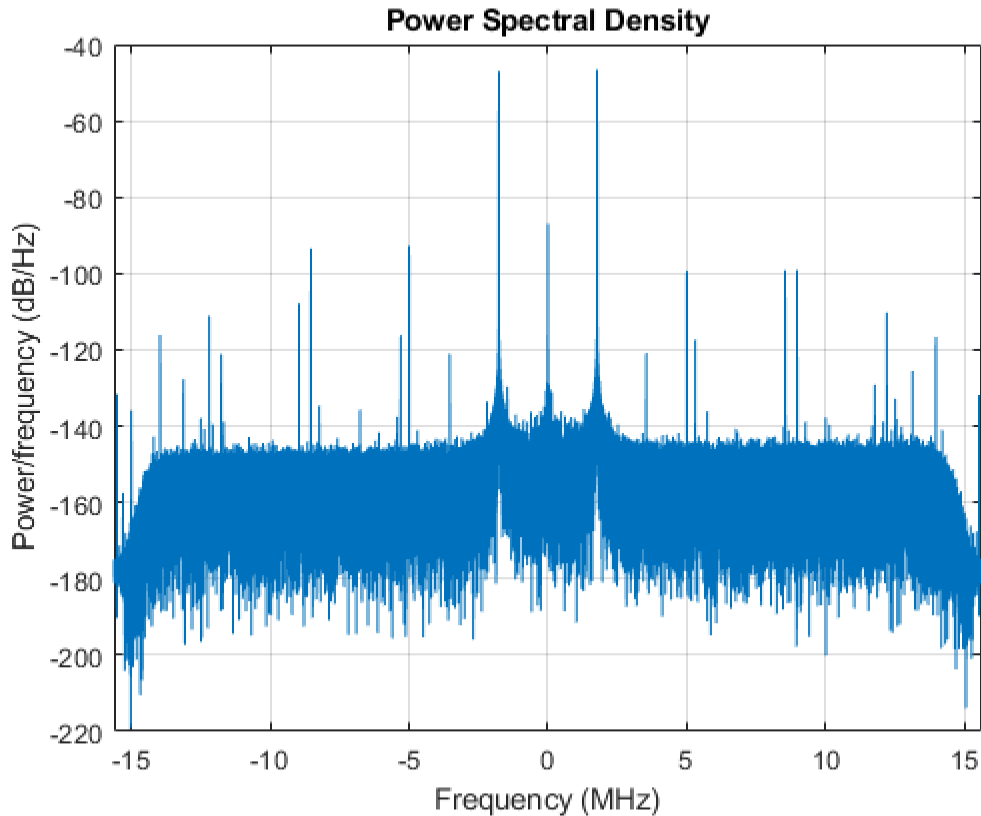
```
hold on
plot(imag(IQData(1:1000)),"g");
legend("Inphase signal", "Quadrature signal");
title("IQ Data for the first 1000 points of acquired signal")
xlabel("Sample number");
ylabel("Voltage");
```



Plot the Spectrum View of the IQ Data

The spectrum view might have more information than the time domain view of the data. For example, you may use the spectrum view to identify the main frequency bands, the signal bandwidth, etc. You need the DSP System Toolbox to plot the spectrum view. You may get errors if the required functionality is not available.

```
% Create a periodogram spectrum with a Hamming window
figure(2)
periodogram(IQData,hamming(length(IQData)),[],sampleRate,"centered")
```



Switch the Instrument Back to Spectrum Analyzer Mode

Switch the instrument to spectrum analyzer mode and compare the spectrum view generated in MATLAB with the view on the Signal Analyzer. Use additional SCPI commands to configure the instrument measurement and display settings.

```
% Switch back to the spectrum analyzer view
writeline(signalAnalyzerObject,":INSTrument:SELect SA");

% Set mechanical attenuation level
writeline(signalAnalyzerObject,":SENSe:POWer:RF:ATTenuation " + num2str(mechAttenuation));

% Set the center frequency, RBW and VBW and trigger
writeline(signalAnalyzerObject,":SENSe:FREQuency:CENTer " + num2str(centerFrequency));
writeline(signalAnalyzerObject,":SENSe:FREQuency:STARt " + num2str(startFrequency));
writeline(signalAnalyzerObject,":SENSe:FREQuency:STOP " + num2str(stopFrequency));
writeline(signalAnalyzerObject,":SENSe:BANDwidth:RESolution " + num2str(resolutionBandwidth));
writeline(signalAnalyzerObject,":SENSe:BANDwidth:VIDeo " + num2str(videoBandwidth));

% Continuous measurement
writeline(signalAnalyzerObject,":INIT:CONT ON");

% Trigger
writeline(signalAnalyzerObject,"*TRG");
```

Clean Up

```
% Close and delete instrument connections
clear signalAnalyzerObject
```

Fetch Waveform through NI-SCOPE MATLAB Instrument Driver in Simulation Mode

This example shows how to acquire digital waveform from two channels of a National Instruments® NI-SCOPE driver in the simulation mode.

Introduction

Instrument Control Toolbox™ supports communication with instruments through high-level drivers. In this example you can acquire digital waveforms from a National Instruments® NI-SCOPE driver in the simulation mode.

Requirements

This example requires a Microsoft® Windows® system and NI-SCOPE package 3.6 or higher. Make sure the Measurement & Automation Explorer recognizes the NI-SCOPE driver before you use this example.

Verify NI-SCOPE Installation

Use the `instrhwinfo` command to check if the NI-SCOPE software package is installed correctly. If installed correctly, NI-SCOPE is listed as one of the modules installed on the Windows machine. This example uses libraries installed with it.

```
driversInfo = instrhwinfo ('ivi');
disp(driversInfo.Modules');
```

```
    {'nidcpower'      }
    {'nidmm'          }
    {'niFgen'         }
    {'nisACPwr'       }
    {'niScope'        }
    {'nisCounter'     }
    {'nisDCPwr'       }
    {'nisDigitizer'   }
    {'nisDmm'         }
    {'nisDownconverter'}
    {'nisFGen'        }
    {'nisPwrMeter'    }
    {'nisRFSigGen'    }
    {'nisScope'       }
    {'nisSpecAn'      }
    {'nisSwtch'       }
    {'nisUpconverter' }
    {'niSwitch'       }
```

Create a MATLAB Instrument object

Use the `icdevice` function to create an instrument object from the MDD you generated, and establish a connection to the scope using that object.

`icdevice` function takes two or more input arguments. The MDD file name, the resource name for the scope and optionally, device-specific parameters that can be set.

You can get the resource name for the scope from NI Measurement and Automation Explorer. For example: A resource name of `PXI1Slot6` in NI MAX would be `PXI1Slot6`. You can remove the `optionstring` argument and the corresponding string parameter if you have the actual hardware.

You can establish a connection to the scope using the `connect` command.

```
ictObj = icdevice('niscope.mdd', 'PXI1Slot6', 'optionstring', 'simulate=true');
connect(ictObj);
disp(ictObj);
```

```
   Instrument Device Object Using Driver : niScope

   Instrument Information
      Type:               IVIInstrument
      Manufacturer:       National Instruments Corp.
      Model:              NI Digitizers

   Driver Information
      DriverType:         MATLAB IVI
      DriverName:         niScope
      DriverVersion:      1.0

   Communication State
      Status:             open
```

Configure the Scope

For the purpose of this example, the scope is configured using auto setup which automatically sets the vertical range, sample rate, trigger level and a few other settings.

Use the MATLAB Instrument Driver Editor (midedit) to view other properties and functions that allow you to configure a device. The tool shows all the properties and functions that the NI-SCOPE software package supports.

```
configuration = ictObj.Configuration;
invoke(configuration, 'autosetup');
```

Configure the vertical range for each channel

ConfigureVertical function configures the most commonly configured attributes of the digitizer vertical subsystem, such as the range, offset, coupling, probe attenuation and the channel. We will be fetching data from channels '0' and '1' and therefore configuring these channels. Refer to the NI-Scope documentation for further information.

```
Range = 10;
Offset = 0;
Coupling = 1;
ProbeAttenuation = 1;

% Configure Channel 0
invoke(ictObj.Configurationfunctionsvertical, 'configurevertical', '0', Range, Offset, Coupling,

% Configure Channel 1
invoke(ictObj.Configurationfunctionsvertical, 'configurevertical', '1', Range, Offset, Coupling,
```

Prepare the waveform information

Waveform information is a structure containing the timing information, number of samples, gain, and offset scaling factors for acquiring a waveform from the scope. Memory will be preallocated for the waveform information structure for each channel.

```
numChannels = 2;
channelList = '0,1';
numSamples = 1024;

for i = 1:numChannels
    waveformInfo(i).absoluteInitialX = 0;
    waveformInfo(i).relativeInitialX = 0;
    waveformInfo(i).xIncrement = 0;
    waveformInfo(i).actualSamples = 0;
    waveformInfo(i).offset = 0;
    waveformInfo(i).gain = 0;
    waveformInfo(i).reserved1 = 0;
    waveformInfo(i).reserved2 = 0;
end
```

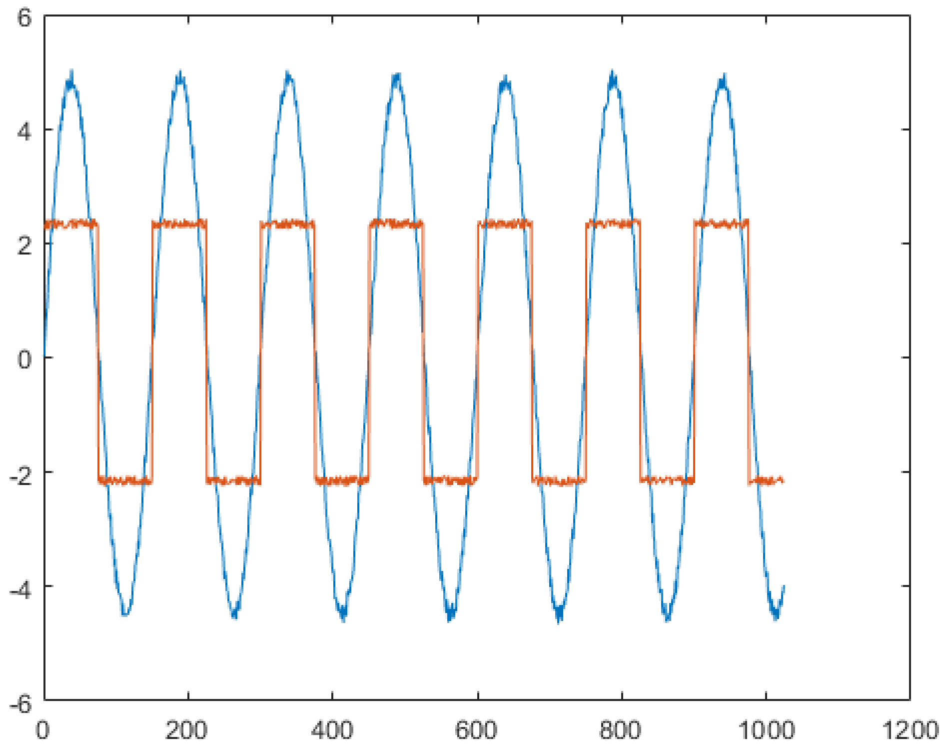
Fetch waveform

Once you configure the scope with the required settings, use an appropriate function call to acquire the waveform from channels `0` and `1`. Preallocate waveformArray to store data from all requested channels. Prior to fetching data, a waveform acquisition has to be initiated. After fetching the data, extract the waveform corresponding to each channel.

```
waveformArray = zeros(numChannels * numSamples, 1);
TimeOut = 10; % seconds
invoke(ictObj.Acquisition, 'initiateacquisition');
[waveformArray, waveformInfo] = invoke(ictObj.Acquisition, 'fetch', channelList,...
    TimeOut, numSamples, waveformArray, waveformInfo);

waveformArray = reshape(waveformArray, numSamples, numChannels);
```

Plot the waveform

```
plot(waveformArray);
```



Clean up

Delete the MATLAB Instrument Object.

```
disconnect(ictObj);
delete(ictObj);
clear ictObj;
```

Using NI-FGEN Instrument Driver To Generate A Sine Wave

This example shows how to generate a sine wave on a function generator using the NI-FGEN software.

Instrument Control Toolbox™ supports communication with instruments through interfaces and drivers.

For a complete list of supported hardware, visit the Instrument Control Toolbox product page.

Requirements

In this example, you will learn to generate a sine wave using the NI-FGEN software package version 2.7.2 or higher and a NI PXI-5402 function generator. You can also use any other function generator supported by the NI-FGEN software package version 2.7.2 or higher.

Verify NI-FGEN Installation

Use the `instrhwinfo` command to check if the NI-FGEN software package is installed correctly.

```
driverInfo = instrhwinfo('ivi');
driverInfo.Modules'
```

```
ans =

  18×1 cell array

    {'AgRfSigGen'      }
    {'nidcpower'       }
    {'nidmm'           }
    {'niFgen'          }
    {'nisACPwr'        }
    {'niScope'         }
    {'nisCounter'      }
    {'nisDCPwr'        }
    {'nisDigitizer'    }
    {'nisDmm'          }
    {'nisDownconverter'}
    {'nisFGen'         }
    {'nisPwrMeter'     }
    {'nisRFSigGen'     }
    {'nisScope'        }
    {'nisSpecAn'       }
    {'nisSwtch'        }
    {'nisUpconverter'  }
```

Create a MATLAB Instrument Object

Use the `icdevice` function to create an instrument object from the MDD, and establish a connection to the function generator using that object.

`icdevice` function takes two or more input arguments. The MDD file name, the resource name for the function generator and optionally, setting specific parameters.

You can get the resource name for the function generator from NI Measurement and Automation tool. For example: A resource name of `PXI1Slot6` in NI MAX would be `DAQ::PXI1Slot6` and `Device 2`

would be DAQ::2. You can remove the optionstring argument and the corresponding string parameter if you have the actual hardware.

```
% Specify Resource ID
resourceID = 'DAQ::PXI1Slot6';
ictObj = icdevice('niFgen',resourceID,'optionstring','Simulate=true,DriverSetup=Model:5402');

% Connect driver instance
connect(ictObj);
```

Attributes and Variables Definition

For the purpose of this example, the function generator is configured to generate a sine wave on Channel 0 with a frequency of 10E6, amplitude of 2, DC Offset of 0 and starting phase of 0.

```
% These values are defined in the driver's header file 'niFgen.h'
NIFGEN_VAL_OUTPUT_FUNC = 0;
NIFGEN_VAL_WFM_SINE = 1;
NIFGEN_ATTR_FUNC_FREQUENCY = 10E+6;
NIFGEN_ATTR_FUNC_AMPLITUDE = 2.0;
NIFGEN_ATTR_FUNC_DC_OFFSET = 0;
NIFGEN_ATTR_FUNC_START_PHASE = 0.0;

% This value is described in the help file 'NI Signal Generators Help'
ChannelName = '0';
```

Configure Output Mode and Waveform

```
invoke(ictObj.Configuration,'configureoutputmode',NIFGEN_VAL_OUTPUT_FUNC);
invoke(ictObj.Configurationfunctionsstandardfunctionoutput,'configurestandardwaveform',ChannelNar
```

Initiate the Waveform Generation

Once you configure the function generator with the required settings, use an appropriate function call to initiate the waveform.

```
invoke(ictObj.Waveformcontrol,'initiategeneration');
```

Enable the Output

Once the waveform generation begins, enable the output of the function generator.

```
Enabled = 1;
invoke(ictObj.Configuration,'configureoutputenabled', ChannelName, Enabled);
```

Clean Up

Delete the MATLAB Instrument Object.

Note: If you delete the MATLAB Instrument Object, it will stop the waveform generation.

```
disconnect(ictObj);
delete(ictObj);
clear ictObj;
```

Read Waveform Data from Keysight® DSO-X 2002A Oscilloscopes Using the IVI-C Driver

This example shows how to initialize the driver, read a few properties of the driver, acquire waveform data using Agilent Technologies DSO-X 2002A oscilloscope and output the result in MATLAB®.

Requirements

This example requires the following to be installed on the computer:

- Keysight (Agilent) IO libraries version 17.1 or newer;
- Keysight (Agilent) 2000, 3000, 4000, 6000 InfiniiVision X-Series Oscilloscope IVI driver version 2.1.6.0 or newer;

Enumerate Available IVI-C Drivers on the Computer

This enumerates the IVI drivers that have been installed on the computer.

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create MATLAB Instrument Driver And Connect To The Instrument

```
% Create the MATLAB instrument driver
makemid('AgInfiniiVision','AgInfiniiVision.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('AgInfiniiVision.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true'

% Connect driver instance
connect(myInstrument);
```

Get General Device Properties

Query information about the driver and instrument

```
% Get information about the driver
Utility = get(myInstrument, 'Utility');
Revision = invoke(Utility, 'revisionquery');
```

```
DriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentification');
InstrumentIdentification = get(myInstrument,'Inherentiviattributesinstrumentidentification');
Vendor = get(DriverIdentification, 'Specific_Driver_Vendor');
Description = get(DriverIdentification, 'Specific_Driver_Description');
InstrumentModel = get(InstrumentIdentification, 'Instrument_Model');
FirmwareRev = get(InstrumentIdentification, 'Instrument_Firmware_Revision');

% Print the queried driver properties
fprintf('Revision:        %s\n', Revision);
fprintf('Vendor:          %s\n', Vendor);
fprintf('Description:     %s\n', Description);
fprintf('InstrumentModel: %s\n', InstrumentModel);
fprintf('FirmwareRev:     %s\n', FirmwareRev);
fprintf(' \n');
```

```
Revision:        2.1.6.0
Vendor:          Agilent Technologies
Description:     IVI driver for the Keysight 2000X, 3000X, 4000X and 6000X family of Oscilloscope
InstrumentModel: DSO-X 2002A
FirmwareRev:     Sim2.1.6.0
```

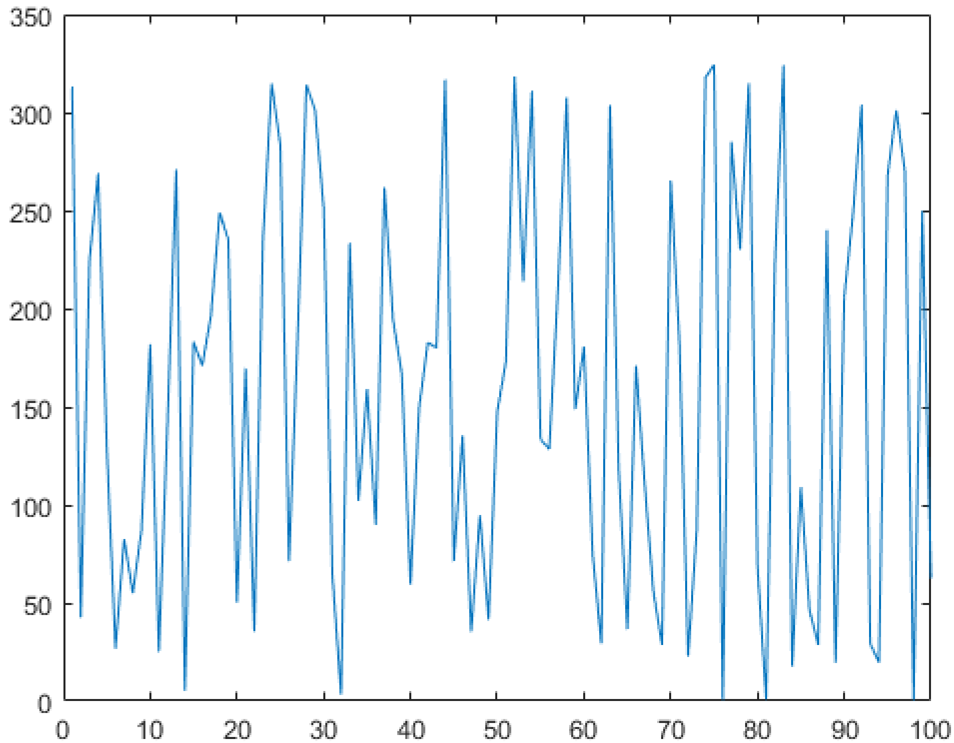
Setup The Measurements And Fetch A Waveform From The Oscilloscope

```
Measurement = get(myInstrument, 'Instrumentspecificmeasurement');
invoke(Measurement, 'measurementsautosetup');
WaveformArray = zeros(1,100);
[WaveformArray,ActualPoints,InitialX,Xincreament] = invoke(Measurement, 'measurementfetchwavefor
```

Visualize Data And Display Any Errors

```
% Display the fetched data
plot(WaveformArray(1:ActualPoints));

% If there are any errors, query the driver to retrieve and display them.
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
ErrorQuery: 0, No error.
```



Disconnect the Device Object and Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory
delete(myInstrument);
```

Additional Information:

This example shows the setup and acquisition of data from an oscilloscope using the IVI driver. Once the measured waveform is retrieved from the instrument, MATLAB can be used to visualize and perform analyses on the data using the rich library of functions in the Signal Processing Toolbox™ and Communications Systems Toolbox™. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Read Waveforms from a Keysight® M9210A Digitizer using the IVI-C Driver

This example shows how to acquire a waveform from both channels of a Keysight technologies M9210A digitizer using an IVI-C driver, and display it in MATLAB®. Instrument Control Toolbox™ software supports communication with instruments through IVI drivers. For a complete list of hardware supported, visit the Instrument Control Toolbox™ supported hardware page.

Introduction

This example has been tested on Microsoft® Windows® 7 and Windows XP Systems. Agilent IO Suite and MD1 IVI driver version 1.2.2.0 need to be installed.

Ensure that the instrument has been configured in the VISA utility (such as Agilent Connection Expert) before you execute this example.

Verify the Driver is Installed

If the AgMD1 driver is installed, it will show up in the list of installedDrivers

```
IviInfo = instrhwinfo('ivi');
installedDrivers = IviInfo.Modules
```

```
installedDrivers =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Import the IVI-C Driver into MATLAB

To use the installed IVI-C driver from MATLAB, a MATLAB driver needs to be created. The MATLAB driver needs to be created only once, and should exist on the MATLAB path

```
% Create the MATLAB driver
makemid('AgMD1', 'AgMD1.mdd', 'ivi-c');
```

Initialize the Driver Object and Connect to the Digitizer

Using the MATLAB driver, a device object must be created first. Using the device object, a connection is established to the digitizer from MATLAB. In this example the driver is used in simulate mode. Set Simulate=false in the initOptions variable below to run this example with an actual instrument

```
initOptions = 'Simulate=true, DriverSetup= Cal=0, Trace=false, model=M9210A';
visaAddress = 'PXI17::10::0::INSTR';
myDigitizer = icdevice('AgMD1.mdd', visaAddress, 'optionstring', initOptions);
```

```
% Connect to the digitizer using the device object created above
connect(myDigitizer);
```

Set Up the Digitizer to Acquire Waveforms on Channel 1 and Channel 2

After the connection is established, properties such as input impedance, number of points per record, and sampling rate need to be set, prior to acquiring the waveforms on the digitizer. The values of the enumerated datatypes used can be found in the MD1 driver documentation

```
% Abort present acquisition if any
invoke(myDigitizer.Waveformacquisitionlowlevelacquisition, 'abort');

% Set the input impedance values of the individual channels
myDigitizer.RepCapIdentifier = 'Channel1';
myDigitizer.Channel.Input_Impedance = 50; % (Ohms)

myDigitizer.RepCapIdentifier = 'Channel2';
myDigitizer.Channel.Input_Impedance = 50; % (Ohms)

% Set the acquisition parameters
numberOfRecords = 1;
ptsPerRecord = 1e4;
samplingRate = 2e9;

invoke(myDigitizer.Configurationacquisition, 'configureacquisition',...
       numberOfRecords, ptsPerRecord, samplingRate);

% Set the individual channel parameters
Range = 0.2;
Offset = 0.0;
Coupling = 1;
Enabled = true;
invoke(myDigitizer.Configurationchannel, 'configurechannel', 'Channel1',...
       Range, Offset, Coupling, Enabled);
invoke(myDigitizer.Configurationchannel, 'configurechannel', 'Channel2',...
       Range, Offset, Coupling, Enabled);

% Set the trigger source, and trigger type
myDigitizer.RepCapIdentifier = 'Channel1';
myDigitizer.Trigger.Active_Trigger_Source = 'External1';

% The hex value can be found in the MD1 driver documentation
AGMD1_VAL_IMMEDIATE_TRIGGER = hex2dec('000003E9');
myDigitizer.Trigger.Trigger_Type = AGMD1_VAL_IMMEDIATE_TRIGGER;

% Determine the minimum amount of memory needed to fetch or read data from
% the digitizer for maximum performance
dataWidth = 64;
numRecords = 1;
offsetWithinRecord = 0;
numPointsPerRecord = ptsPerRecord;

arrayElements = invoke(myDigitizer.Waveformacquisitionlowlevelacquisition,...
                       'queryminwaveformmemory', dataWidth, numRecords,...
                       offsetWithinRecord, numPointsPerRecord);
WaveformArray = zeros(arrayElements, 1);
```

Acquire Data

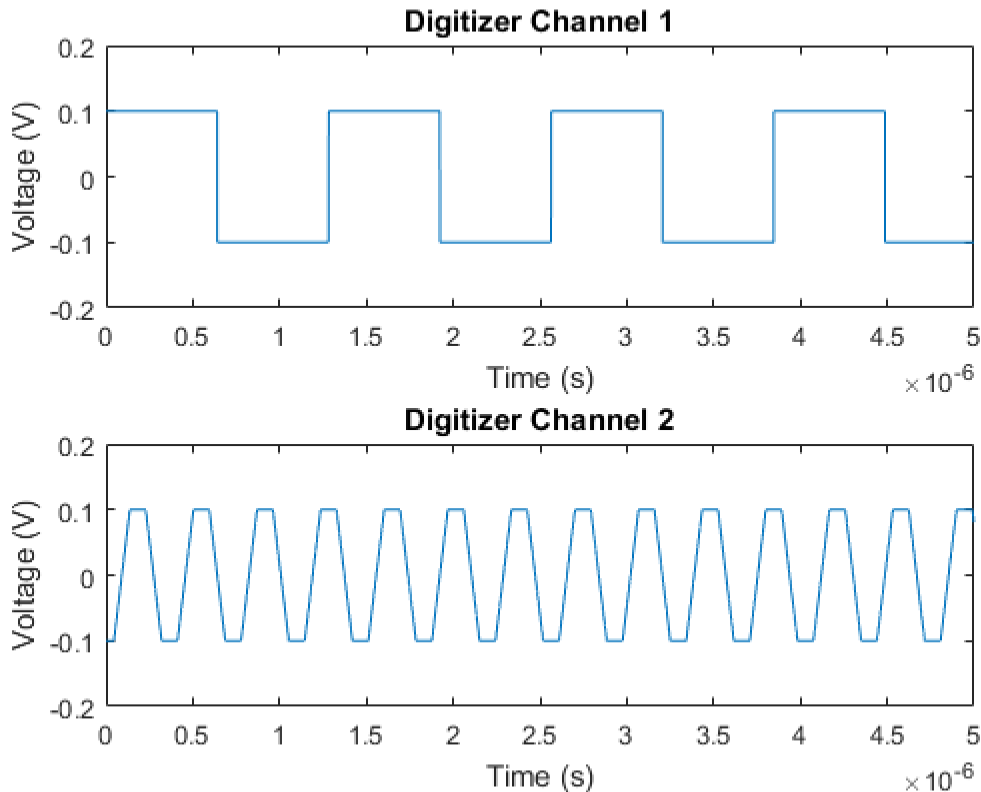
To acquire data from channel 1 the READWAVEFORMREAL64 method is used. The READWAVEFORMREAL64 method initiates acquisition of a signal on both channel 1 and channel 2, but returns the waveform for channel 1 only. Following this, to read the already acquired waveform from channel 2, the FETCHWAVEFORMREAL64 method is used

```
maxTimeMilliseconds = 50;

% Initiate an acquisition on all enabled channels, wait (up to
% |maxTimeMilliseconds|) for the acquisition to complete, and return the
% waveform for this channel

[sig1, ActualPoints, FirstValidPoint, ~, ~, ~, XIncrement] = ...
    invoke(myDigitizer.Waveformacquisition, 'readwaveformreal64', ...
    'Channel1', maxTimeMilliseconds, arrayElements, WaveformArray);
sig1 = sig1(FirstValidPoint+1:FirstValidPoint+ActualPoints);

% |FETCHWAVEFORMREAL64| returns a previously acquired waveform for the selected
% channel. An acquisition must be made prior to calling this method. For
% this case the previous call to |READWAVEFORMREAL64| has performed the waveform
% acquisition already. Call this method separately for each channel

[sig2, ActualPoints, FirstValidPoint, ~, ~, ~, ~] = ...
    invoke(myDigitizer.Waveformacquisitionlowlevelacquisition, ...
    'fetchwaveformreal64', 'Channel2', arrayElements, WaveformArray);
sig2 = sig2(FirstValidPoint+1:FirstValidPoint+ActualPoints);
```

Plot the acquired signals

The SUBPLOT feature of MATLAB is used to plot the waveforms read from channel 1 and channel 2 of the digitizer

```
% Create the time vector using the XIncrement value returned by the
% |READWAVEFORMREAL64| method
t = 0:XIncrement:(length(sig1)-1)*XIncrement;

figure;
% Plot the channel 1 waveform
subplot(2, 1, 1);
plot(t, sig1);
xlabel('Time (s)');
ylabel('Voltage (V)');
ylim([-.2 0.2]);
title('Digitizer Channel 1');

% Plot the channel 2 waveform
subplot(2, 1, 2);
plot(t, sig2);
xlabel('Time (s)');
ylabel('Voltage (V)');
ylim([-.2 0.2]);
title('Digitizer Channel 2');
```



Clean Up - Delete the MD1 Device Object

After configuring the digitizer, and fetching/reading data from it, the device object needs to be closed and removed from the workspace

```
disconnect(myDigitizer);
delete(myDigitizer);
clear myDigitizer;
```

Additional Information:

This example shows the setup and acquisition of data from a Digitizer using the IVI driver. Once the measured waveform is retrieved from the instrument, MATLAB can be used to visualize and perform analyses on the data using the rich library of functions in the Signal Processing Toolbox™ and Communications Systems Toolbox™. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Set Output Voltage and Make Measurements on Keysight® AC6801A Power Supply Using the IVI-C Driver

This example shows the use of MATLAB® with an IVI driver to connect to, configure and measure AC voltage using Keysight Technologies AC6801A power supply and output the result in MATLAB.

Requirements

This example requires the following to be installed on the computer:

- Keysight IO libraries version 17.1 or newer
- Keysight AC6800 AC Power Supplies IVI version 1.0.3.0 or newer

Enumerate available IVI-C drivers on the computer

This enumerates the IVI drivers that have been installed on the computer.

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create MATLAB Instrument Driver and Connect to the Instrument

```
% Create the MATLAB instrument driver
makemid('AgAC6800','AgAC6800.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('AgAC6800.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true');

% Connect driver instance
connect(myInstrument);
```

Attributes Definition and Variables Definition

```
% Attributes Definition. These values are defined in the driver's header file, 'AgAC6800.h'
IVI_ATTR_BASE= 1000000;
IVI_SPECIFIC_ATTR_BASE = IVI_ATTR_BASE + 150000;
AGAC6800_ATTR_OUTPUT_PHASE_VOLTAGE_SOFT_LIMIT_ENABLED = IVI_SPECIFIC_ATTR_BASE + 15;
AGAC6800_VAL_MODE_AC_DC = 2;
AGAC6800_ATTR_PROTECTION_CURRENT_PROTECTION_ENABLED = IVI_SPECIFIC_ATTR_BASE + 37;
```

```
AGAC6800_VAL_MEASUREMENT_TYPE_VOLTAGERMS = 0;
AGAC6800_VAL_MEASUREMENT_TYPE_CURRENTRMS = 1;
AGAC6800_VAL_MEASUREMENT_TYPE_VOLTAGEDC = 3;
AGAC6800_VAL_MEASUREMENT_TYPE_CURRENTDC = 4;
AGAC6800_VAL_MEASUREMENT_TYPE_POWER_FACTOR = 5;
AGAC6800_VAL_MEASUREMENT_TYPE_CREST_FACTOR = 6;
AGAC6800_VAL_MEASUREMENT_TYPE_CURRENT_PEAK = 7;
AGAC6800_VAL_MEASUREMENT_TYPE_POWERVA = 8;
AGAC6800_VAL_MEASUREMENT_TYPE_POWERDC = 9;
AGAC6800_VAL_MEASUREMENT_TYPE_POWER_REACTIVE = 12;
AGAC6800_VAL_MEASUREMENT_TYPE_VOLTAGEACDC = 13;
AGAC6800_VAL_MEASUREMENT_TYPE_CURRENTACDC = 14;
AGAC6800_VAL_MEASUREMENT_TYPE_POWER_REAL = 19;
AGAC6800_VAL_MEASUREMENT_TYPE_POWER_REALACDC = 15;
AGAC6800_VAL_MEASUREMENT_TYPE_POWERVAACDC = 16;
AGAC6800_VAL_MEASUREMENT_TYPE_POWER_REACTIVEACDC = 17;
AGAC6800_VAL_MEASUREMENT_TYPE_POWER_FACTORACDC = 18;

% Variables Definition. These values are being passed to the driver.
AcVolts = 120;
AcAmps = 4;
Freq = 50;
LoVolt = 0.9 * AcVolts;
HiVolt = 1.1 * AcVolts;
DcVolts = 5;
DcAmps = 1;
```

Get General Device Properties

Query information about the driver and instrument

```
AttributeAccessors = get(myInstrument, 'Attributeaccessors');
DriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentification');
DCGeneration = get(myInstrument,'Dcgeneration');
InstrumentIdentification = get(myInstrument,'Inherentiviattributesinstrumentidentification');
InstrumentModel = get(InstrumentIdentification, 'Instrument_Model');
InstrumentSpecificMeasurement = get(myInstrument, 'Instrumentspecificmeasurement');
InstrumentSpecificOutputPhaseCurrent = get(myInstrument,'Instrumentspecificoutputphasecurrent');
InstrumentSpecificOutputPhaseFrequency = get(myInstrument, 'Instrumentspecificoutputphasefrequen
InstrumentSpecificOutputPhaseVoltage = get(myInstrument,'Instrumentspecificoutputphasevoltage');
InstrumentSpecificOutputPhaseVoltageSoftLimit = get(myInstrument,'Instrumentspecificoutputphasev
InstrumentSpecificSystem = get(myInstrument, 'Instrumentspecificsystem');
Output = get(myInstrument,'Output');
Outputs = get(myInstrument,'Outputs');
Utility = get(myInstrument, 'Utility');

Revision = invoke(Utility, 'revisionquery');
Vendor = get(DriverIdentification, 'Specific_Driver_Vendor');
Description = get(DriverIdentification, 'Specific_Driver_Description');
FirmwareRev = get(InstrumentIdentification, 'Instrument_Firmware_Revision');

% Print the queried driver properties
fprintf('Revision:        %s\n', Revision);
fprintf('Vendor:          %s\n', Vendor);
fprintf('Description:     %s\n', Description);
fprintf('InstrumentModel: %s\n', InstrumentModel);
fprintf('FirmwareRev:     %s\n', FirmwareRev);
fprintf(' \n');
```

```
Revision:        1.0.3.0
Vendor:          Agilent Technologies
Description:     IVI Driver for AC68xx family of Power Supplies [Compiled for 64-bit.]
InstrumentModel: AC6801A
FirmwareRev:     Sim1.0.3.0
```

Enable Auto-Ranging and Configure Output AC Voltage, Current and Frequency

```
% Enable the voltage auto range so that the maximum current will be
% supplied for the voltage setting
set(InstrumentSpecificOutputPhaseVoltage, 'Auto_Range', true);

% Set voltage level to 120 V
set(Output, 'Voltage_Level', AcVolts);

% Set output phase current limit to 4 A
invoke(Output, 'configurecurrentlimit', 'OutputPhase1',AcAmps);

% Set output phase frequency level to 50 Hz
invoke(Outputs, 'configurefrequency', 50);
```

Configure Phase Voltage Soft Limit

The AC sources also allow you to set some protection features. The first one is soft limits. For this example, the limits will be +/- 10%.

```
% Set output phase voltage soft limit lower limit to 108 V
set(InstrumentSpecificOutputPhaseVoltageSoftLimit, 'Lower_Limit', LoVolt);

% Set output phase voltage soft limit upper limit to 132 V
set(InstrumentSpecificOutputPhaseVoltageSoftLimit, 'Upper_Limit', HiVolt);

% Enable the output phase voltage soft limit
invoke(AttributeAccessors, 'setattributeviboolean', 'OutputPhase1', AGAC6800_ATTR_OUTPUT_PHASE_VO
```

Configure Output DC Voltage And Current

```
% Set output voltage generation mode to AC Plus DC. Set the DC voltage level to 5 V
invoke(DCGeneration, 'configuredc', 'OutputPhase1',AGAC6800_VAL_MODE_AC_DC,DcVolts);

% Set the output phase current DC limit to 1 A
set(InstrumentSpecificOutputPhaseCurrent, 'Dc_Limit', DcAmps);
```

Enable Current Protection Feature and Turn the Output On

There is also current protection to enable. The instrument will go into protect when it hits current limit

```
% Enable the current protection
invoke(AttributeAccessors, 'setattributeviboolean', '', AGAC6800_ATTR_PROTECTION_CURRENT_PROTECTI

% Enable output
set(Output, 'Enabled', 1);

% Wait for operation to complete.
invoke(InstrumentSpecificSystem, 'systemwaitforoperationcomplete', 1000); % Wait for 1000ms maxim
```

Make Voltage Measurements

The AC Source can take a bunch of measurements. Once you do one, you can fetch the rest from the same acquisition

```
fprintf('Voltage Measurements\n');

VoltAC = invoke(InstrumentSpecificMeasurement, 'measurementmeasure', 'Measurement1', AGAC6800_VAI
fprintf('Measured AC Voltage %0.15g V\n', VoltAC);

VoltDC = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAL_I
fprintf('Fetched DC Voltage = %0.15g V\n', VoltDC);

VoltACDC = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAI
fprintf('Fetched AC+DC Voltage = %0.15g V rms\n', VoltACDC);
```

```
Voltage Measurements
Measured AC Voltage 0 V
Fetched DC Voltage = 0 V
Fetched AC+DC Voltage = 0 V rms
```

Make Current Measurements

```
fprintf('\nCurrent Measurements\n');

CurrAC = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAL_I
fprintf('Fetched AC Current = %0.15g A rms\n', CurrAC);

CurrDC = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAL_I
fprintf('Fetched DC Current = %0.15g A\n', CurrDC);

CurrACDC = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAI
fprintf('Fetched AC+DC Current = %0.15g A rms\n', CurrACDC);

PeakCurr = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAI
fprintf('Fetched Peak Current = %0.15g A\n', PeakCurr);

CresFact = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAI
fprintf('Fetched Crest Factor = %0.15g\n', CresFact);
```

```
Current Measurements
Fetched AC Current = 0 A rms
Fetched DC Current = 0 A
Fetched AC+DC Current = 0 A rms
Fetched Peak Current = 0 A
Fetched Crest Factor = 0
```

Make Power Measurements

```
fprintf('\nPower Measurements\n');

RealPow = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAL_
fprintf('Fetched AC Real Power = %0.15g W\n', RealPow);

ApparPow = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAI
fprintf('Fetched AC Apparent Power = %0.15g VA\n', ApparPow);

ReactPow = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAI
```

```
fprintf('Fetched AC Reactive Power = %0.15g VAR\n', ReactPow);

DCPow = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAL_MI
fprintf('Fetched DC Power = %0.15g W\n', DCPow);

PowFact = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_VAL_
fprintf('Fetched Power Factor = %0.15g\n', PowFact);

RealPowACDC = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_
fprintf('Fetched AC+DC Real Power = %0.15g W\n', RealPowACDC);

ApparPowACDC = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800
fprintf('Fetched AC+DC Apparent Power = %0.15g VA\n', ApparPowACDC);

ReactPowACDC = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800
fprintf('Fetched AC+DC Reactive Power = %0.15g VAR\n', ReactPowACDC);

PowFactACDC = invoke(InstrumentSpecificMeasurement, 'measurementfetch', 'Measurement1', AGAC6800_
fprintf('Fetched AC+DC Power Factor = %0.15g\n', PowFactACDC);
fprintf('\n');
```

```
Power Measurements
Fetched AC Real Power = 0 W
Fetched AC Apparent Power = 0 VA
Fetched AC Reactive Power = 0 VAR
Fetched DC Power = 0 W
Fetched Power Factor = 0
Fetched AC+DC Real Power = 0 W
Fetched AC+DC Apparent Power = 0 VA
Fetched AC+DC Reactive Power = 0 VAR
Fetched AC+DC Power Factor = 0
```

Query and Display any Errors

```
% If there are any errors, query the driver to retrieve and display them.
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
ErrorQuery: 0, No error.
```

Disconnect Device Object And Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory.
delete(myInstrument);
```

Additional Information:

This example shows setting output voltage and make power measurements from a power supply using the IVI driver. Once the measured power data is retrieved from the instrument, MATLAB can be used to visualize and perform analyses on the data using the rich library of functions in the Signal Processing Toolbox™ and Communications Systems Toolbox™. Using Instrument Control Toolbox™, it

is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Measure Frequency on Keysight® 532xx Frequency Counter Using the IVI-C Driver

This example shows how to initialize the driver, read a few properties of the driver and measure frequency using Keysight Technologies 532xx Frequency Counter and output the result in MATLAB®.

Requirements

This example requires the following:

- Keysight IO libraries version 17.1 or newer
- Keysight 532xx Frequency Counter IVI driver version 1.0.11.0 or newer

Enumerate Available IVI-C Drivers On The Computer

This enumerates the IVI drivers that have been installed on the computer.

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create MATLAB Instrument Driver And Connect To The Instrument

```
% Create the MATLAB instrument driver
makemid('Ag532xx','Ag532xx.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('Ag532xx.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true');

% Connect driver instance
connect(myInstrument);
```

Attributes and Variables Definition

```
% These values are defined in the driver's header file 'Ag532xx.h'
AG532XX_VAL_SLOPE_POSITIVE = 1;
AG532XX_VAL_TRIGGER_SOURCE_IMMEDIATE = 0;
AG532XX_VAL_ARM_MEASUREMENT_TYPE_FREQUENCY = 0;
AG532XX_VAL_ARM_SOURCE_ADVANCED = 3;
```

```
AG532XX_VAL_IMMEDIATE_ARM_TYPE = 1;
AG532XX_VAL_DELAY_HOLD_OFF_SOURCE_TIME = 1;
```

Get General Device Properties

Query information about the driver and instrument

```
InherentIviAttributesDriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentifi
InherentIviAttributesInstrumentIdentification = get(myInstrument,'Inherentiviattributesinstrumen
Utility = get(myInstrument, 'Utility');
Revision = invoke(Utility, 'revisionquery');
Vendor = get(InherentIviAttributesDriverIdentification, 'Specific_Driver_Vendor');
Description = get(InherentIviAttributesDriverIdentification, 'Specific_Driver_Description');
InstrumentModel = get(InherentIviAttributesInstrumentIdentification, 'Instrument_Model');
FirmwareRev = get(InherentIviAttributesInstrumentIdentification, 'Instrument_Firmware_Revision')

% Print the queried driver properties
fprintf('Revision:        %s\n', Revision);
fprintf('Vendor:          %s\n', Vendor);
fprintf('Description:     %s\n', Description);
fprintf('InstrumentModel: %s\n', InstrumentModel);
fprintf('FirmwareRev:     %s\n', FirmwareRev);
fprintf(' \n');
```

```
Revision:        1.0.11.0
Vendor:          Agilent Technologies
Description:     IVI Driver for Agilent 532xx family of Counters. [Compiled for 64-bit.]
InstrumentModel: 53230A
FirmwareRev:     Sim1.0.11.0
```

Basic Frequency Measurement

```
fprintf('\nBasic Frequency Measurement \n');
% Configure the counter for a frequency measurement on the specified
% channel
InstrumentSpecificFrequency = get(myInstrument,'Instrumentspecificfrequency');
invoke(InstrumentSpecificFrequency, 'frequencyconfigure', 1E6,1E-4,1);
% Sets the number of triggers that will be accepted by the instrument
% before returning to the "idle" trigger state
InstrumentSpecificTrigger = get(myInstrument,'Instrumentspecifictrigger');
set(InstrumentSpecificTrigger, 'Trigger_Count', 1);
% Sets trigger source for measurements.
% IMMediate (continuous) source immediately issues the trigger
set(InstrumentSpecificTrigger, 'Trigger_Source', AG532XX_VAL_TRIGGER_SOURCE_IMMEDIATE);
% Sets slope of the trigger signal on the rear-panel Trig In BNC
set(InstrumentSpecificTrigger, 'Trigger_Slope', AG532XX_VAL_SLOPE_POSITIVE);
% Initiates a measurement based on the current configuration.
MeasurementLowLevelMeasurement = get(myInstrument,'Measurementlowlevelmeasurement');
invoke(MeasurementLowLevelMeasurement, 'initiate');
% Does not return until previously started operations complete or more
% MaxTimeMilliseconds milliseconds of time have expired.
InstrumentSpecificSystem = get(myInstrument,'Instrumentspecificsystem');
invoke(InstrumentSpecificSystem, 'systemwaitforoperationcomplete', 10000);
% Retrieves the result from a previously initiated measurement.
RetVal = invoke(MeasurementLowLevelMeasurement, 'fetch');
fprintf('Data: %0.15g\n', RetVal);
```

```
Basic Frequency Measurement
Data: 0
```

Measure Frequency With Averaging Function

Gets the mathematical average (mean) of all measurements taken since the last time statistics were cleared. The purpose of using averaging function is to increase the reliability and signal-to-noise ratio of the measurements.

```
fprintf('\nAveraging  \n');
% Configures the counter for a frequency measurement on the specified
% channel
invoke(InstrumentSpecificFrequency, 'frequencyconfigure', 1E6,1E-4,1);
% Sets the number of triggers that will be accepted by the instrument
% before returning to the "idle" trigger state
set(InstrumentSpecificTrigger, 'Trigger_Count', 1);
% Sets trigger source for measurements.
% Immediate (continuous) source immediately issues the trigger
set(InstrumentSpecificTrigger, 'Trigger_Source', AG532XX_VAL_TRIGGER_SOURCE_IMMEDIATE);
% Sets the number of measurements samples the instrument will take per trigger
set(InstrumentSpecificTrigger, 'Trigger_Sample_Count', 3);
% Sets slope of the trigger signal on the rear-panel Trig In BNC
set(InstrumentSpecificTrigger, 'Trigger_Slope', AG532XX_VAL_SLOPE_POSITIVE);
% Enables all calculate commands
InstrumentSpecificMath = get(myInstrument, 'Instrumentspecificmath');
set(InstrumentSpecificMath, 'Math_Enabled', true);
% Enables statistics computation
InstrumentSpecificMathStatistics = get(myInstrument,'Instrumentspecificmathstatistics');
set(InstrumentSpecificMathStatistics, 'Statistics_Enabled', true);
% Initiates a measurement based on the current configuration.
invoke(MeasurementLowLevelMeasurement, 'initiate');
% Does not return until previously started operations complete or more
% MaxTimeMilliseconds milliseconds of time have expired.
invoke(InstrumentSpecificSystem, 'systemwaitforoperationcomplete', 10000);
% Gets the mathematical average (mean) of all measurements taken since
% the last time statistics were cleared
Average = get(InstrumentSpecificMathStatistics, 'Statistics_Average');
fprintf('Average : %0.15g \n',  Average);
```

```
Averaging
Average : 1000000
```

Measure Time Stamp

Time stamp measurements record frequency signal edges as they occur on the counter input channels

```
fprintf('\nTime Stamp Measurement \n');
% Set all measurement parameters and trigger parameters to default
% values selectively for timestamp measurements, then immediately
% trigger a measurement.
InstrumentSpecificTimeStamp = get(myInstrument,'Instrumentspecifictimestamp');
% [VAL,VALACTUALSIZE] = INVOKE(OBJ,'timestampmeasure',COUNT,CHANNEL,VALBUFFERSIZE,VAL)
[Val, ValActualSize] = invoke(InstrumentSpecificTimeStamp, 'timestampmeasure', 10,1,10,zeros(10)
fprintf('%d Data Points: \n',ValActualSize);
for i= 1:ValActualSize
```

```
    fprintf('%0.15g\t\n', Val(i));
end
```

```
Time Stamp Measurement
10 Data Points:
0
1e-06
2e-06
3e-06
4e-06
5e-06
6e-06
7e-06
8e-06
9e-06
```

Measure Time Interval Between Two Channels

The measurement starts on a positive (rising) edge on channel 1, and stops on a positive edge on channel 2.

```
fprintf('\nTime Intervals Measurement with delay \n');
% Configures instrument for a time interval measurement on the specified
% channels
InstrumentSpecificTimeInterval = get(myInstrument,'Instrumentspecifictimeinterval');
invoke(InstrumentSpecificTimeInterval, 'timeintervalconfigure', 1,2);
% Set the gate source for frequency measurement
InstrumentSpecificArm = get(myInstrument,'Instrumentspecificarm');
invoke(InstrumentSpecificArm, 'armsetsource', AG532XX_VAL_ARM_MEASUREMENT_TYPE_FREQUENCY,AG532XX_
% Configures the Start Arm for armed measurements.
Configuration = get(myInstrument,'Configuration');
invoke(Configuration, 'configurestartarm', AG532XX_VAL_IMMEDIATE_ARM_TYPE);
% Set the source used to delay the arm start
InstrumentSpecificArmStartExternal = get(myInstrument,'Instrumentspecificarmstartexternal');
set(InstrumentSpecificArmStartExternal,'Start_External_Delay_Source', AG532XX_VAL_DELAY_HOLD_OFF_
% Set the delay, in seconds, used after an external armed measurement
% has been armed.
Arming = get(myInstrument,'Arming');
set(Arming, 'External_Start_Arm_Delay', 0.1);
% Configure the Stop Arm for armed measurements.
invoke(Configuration, 'configurestoparm', AG532XX_VAL_IMMEDIATE_ARM_TYPE);
% Set the source used to hold off enabling the stop Arm source
InstrumentSpecificArmStopExternal = get(myInstrument,'Instrumentspecificarmstopexternal');
set(InstrumentSpecificArmStopExternal, 'Stop_External_Hold_Off_Source', AG532XX_VAL_DELAY_HOLD_O
% Specify the delay, in seconds, after the External Arm Stop event
% has occurred until the measurement stops.
set(Arming, 'External_Stop_Arm_Delay', 0.25);
% Sets the number of triggers that will be accepted by the instrument
% before returning to the "idle" trigger state
set(InstrumentSpecificTrigger, 'Trigger_Count', 1);
% Sets the number of measurements samples the instrument will take per trigger
set(InstrumentSpecificTrigger, 'Trigger_Sample_Count', 3);
% Initiates a measurement based on the current configuration.
invoke(MeasurementLowLevelMeasurement, 'initiate');
% Wait until initialization operation complete or 50000 milliseconds
% of time have expired
invoke(InstrumentSpecificSystem, 'systemwaitforoperationcomplete', 50000);
```

```
% Retrieves the result from a previously initiated measurement.
RetVal = invoke(MeasurementLowLevelMeasurement, 'fetch');
fprintf('Data: %0.15g\n', RetVal);
fprintf('\n');
```

```
Time Intervals Measurement with delay
Data: 0
```

Display Any Errors From The Driver

```
% If there are any errors, query the driver to retrieve and display them.
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
ErrorQuery: 0, No error.
```

Disconnect Device Object and Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory.
delete(myInstrument);
```

Additional Information:

This example shows the setup and acquisition of frequency from a Counter using the IVI driver. Once the measured frequency is retrieved from the instrument, MATLAB can be used to visualize and perform analyses on the data using the rich library of functions in the Signal Processing Toolbox™ and Communications Systems Toolbox™. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Measure AC Voltage on a Keysight® 34410A Digital Multimeter Using the IVI-C Driver

This example shows how to initialize the driver, read a few properties of the driver, measure AC voltage using Agilent Technologies 34410A digital multimeter and output the result in MATLAB®.

Requirements

This example requires the following:

- Keysight IO libraries version 17.1
- Keysight 34410A Digital Multimeter IVI driver version 1.1.0.0

Enumerate Available IVI-C Drivers on the Computer

This enumerates the IVI drivers that have been installed on the computer

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create a MATLAB Instrument Driver and Connect to the Instrument

```
% Create the MATLAB instrument driver
makemid('Ag34410','Ag34410.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('Ag34410.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true');

% Connect driver instance
connect(myInstrument);
```

Attributes and Variables Definition

```
% These values are defined in the driver's header file 'Ag34410.h'
AG34410_VAL_AC_VOLTS = 2;
AG34410_VAL_IMMEDIATE = 1;
```

Get General Device Properties

Query information about the driver and instrument

```
% Get information about the driver
Utility = get(myInstrument, 'Utility');
Revision = invoke(Utility, 'revisionquery');
DriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentification');
InstrumentIdentification = get(myInstrument,'Inherentiviattributesinstrumentidentification');
Vendor = get(DriverIdentification, 'Specific_Driver_Vendor');
Description = get(DriverIdentification, 'Specific_Driver_Description');
InstrumentModel = get(InstrumentIdentification, 'Instrument_Model');
FirmwareRev = get(InstrumentIdentification, 'Instrument_Firmware_Revision');

% Print the queried driver properties
disp(['Revision:        ', Revision]);
disp(['Vendor:          ', Vendor]);
disp(['Description:     ', Description]);
disp(['InstrumentModel: ', InstrumentModel]);
disp(['FirmwareRev:     ', FirmwareRev]);
fprintf('\n');
```

```
Revision:        1.1.0.0
Vendor:          Agilent Technologies
Description:     IVI driver for the Agilent 34410/11 family of digital multimeters. [Compiled fo
InstrumentModel: 34410A
FirmwareRev:     Sim1.1.0.0
```

Configure the AC Voltage Measurement

```
AutoRange = -1;
Resolution = 0.001;
ConfigGroup = get(myInstrument, 'Configuration');
invoke(ConfigGroup, 'configuremeasurement', AG34410_VAL_AC_VOLTS, AutoRange, Resolution);
```

Configure Multipoint Acquisition

```
TriggerCount = 1;
SampleCount = 10;
SampleInterval = 0.0001;
ConfigMultipoint = get(myInstrument, 'Configurationmultipoint');
invoke(ConfigMultipoint, 'configuremultipoint', TriggerCount,SampleCount,AG34410_VAL_IMMEDIATE,Sa
```

Initiate the Low Level Measurement

Call the `INITIATE` function

```
LowLevelMeasurement = get(myInstrument, 'Measurementlowlevelmeasurement');
invoke(LowLevelMeasurement, 'initiate');
fprintf('Measuring AC Volts\n');
```

```
Measuring AC Volts
```

Fetch Data

```
ReadingArray = zeros(1,10);
[ReadingArray,actualPoints] = invoke(LowLevelMeasurement, 'fetchmultipoint', 5000,10,ReadingArray
```

Visualize Data and Display any Errors

```
% Display the fetched data
fprintf('Measured Data: ');
fprintf('%d ',ReadingArray(1:actualPoints));
fprintf('\n\n');

% If there are any errors, query the driver to retrieve and display them.
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
Measured Data: 0 0 0

ErrorQuery: 0, No error.
```

Disconnect Device Object and Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory
delete(myInstrument);
```

Additional Information:

This example shows the setup and making measurements from a digital multimeter using the IVI driver. Once the measured data is retrieved from the instrument, MATLAB can be used to visualize and perform analyses on the data using the rich library of functions in the Signal Processing Toolbox™ and Communications Systems Toolbox™. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Set Output Voltage and Make Measurements from a Keysight® AgE3633A DC Power Supply Using the IVI-C Driver

This example shows how to initialize the driver, read a few properties of the driver, set output voltage, enable all outputs and measure the output voltage, disable all outputs and measure the output voltage by using Keysight Technologies E3633A DC power supply and output the result in MATLAB®.

Requirements

This example requires the following:

- Keysight IO libraries version 17.1 or newer
- Keysight E36xx DC Power Supply IVI version 1.2.0.0 or newer

Enumerate Available IVI-C Drivers on the Computer

This enumerates the IVI drivers that have been installed on the computer.

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create MATLAB Instrument Driver And Connect to the Instrument

```
% Create the MATLAB instrument driver
makemid('AgE36xx','AgE3633A.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('AgE3633A.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true');

% Connect driver instance
connect(myInstrument);
```

Attributes and Variables Definition

```
% These values are defined in the driver's header file 'AgE36xx.h'
IVI_ATTR_BASE = 1000000;
IVI_CLASS_ATTR_BASE = IVI_ATTR_BASE + 250000;
AGE36XX_ATTR_VOLTAGE_LEVEL = IVI_CLASS_ATTR_BASE + 1;
AGE36XX_VAL_MEASURE_VOLTAGE = 1;
```

Get General Device Properties

Query information about the driver and instrument

```
DriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentification');
InstrumentIdentification = get(myInstrument,'Inherentiviattributesinstrumentidentification');
Utility = get(myInstrument, 'Utility');
Revision = invoke(Utility, 'revisionquery');
Vendor = get(DriverIdentification, 'Specific_Driver_Vendor');
Description = get(DriverIdentification, 'Specific_Driver_Description');
InstrumentModel = get(InstrumentIdentification, 'Instrument_Model');
FirmwareRev = get(InstrumentIdentification, 'Instrument_Firmware_Revision');

% Print the queried driver properties
fprintf('Revision:        %s\n', Revision);
fprintf('Vendor:          %s\n', Vendor);
fprintf('Description:     %s\n', Description);
fprintf('InstrumentModel: %s\n', InstrumentModel);
fprintf('FirmwareRev:     %s\n', FirmwareRev);
fprintf(' \n');
```

```
Revision:        1.2.1.0
Vendor:          Agilent Technologies
Description:     IVI driver for the Agilent E36xx family of programmable power supplies [Compiled
InstrumentModel: E3633A
FirmwareRev:     Sim1.2.1.0
```

Set Output Voltage

```
Configuration = get(myInstrument, 'Configuration');
invoke(Configuration, 'configurevoltagelevel', 'Output1', 1.23);
fprintf('Output 1 set to: 1.23 Volts \n');
```

```
Output 1 set to: 1.23 Volts
```

Enable All Outputs and Measure the Output Voltage

```
Outputs = get(myInstrument, 'Outputs');
set(Outputs, 'Enabled', true);
fprintf('All outputs enabled \n');

% The measured voltage should read as the set value.
Action = get(myInstrument, 'Action');
MeasVal = invoke(Action, 'measure', 'Output1', AGE36XX_VAL_MEASURE_VOLTAGE);
fprintf('Output 1 Measurement = %.4g Volts\n', MeasVal);
```

```
All outputs enabled
Output 1 Measurement = 1.23 Volts
```

Disable All Outputs and Measure the Output Voltage

```
set(Outputs, 'Enabled', false);
fprintf('All outputs disabled\n');

% Since all outputs have been disabled, the measured voltage should be 0.
MeasVal = invoke(Action, 'measure', 'Output1', AGE36XX_VAL_MEASURE_VOLTAGE);
fprintf('Output 1 Measurement = %.4g Volts\n\n', MeasVal);
```

```
All outputs disabled
Output 1 Measurement = 0 Volts
```

Display Any Errors from the Driver

```
% If there are any errors, query the driver to retrieve and display them.
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
ErrorQuery: 0, No error.
```

Disconnect the Device Object and Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory.
delete(myInstrument);
```

Additional Information:

This example shows setting output voltage and making measurements on a DC power supply using the IVI driver. Once the measured voltage is retrieved from the instrument, MATLAB can be used to visualize and perform analyses on the data using the rich library of functions in the Signal Processing Toolbox™ and Communications Systems Toolbox™. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Generate AM Waveforms on Keysight® 3352x Waveform Generator Using the IVI-C Driver

This example shows how to initialize the driver, read a few properties of the driver, generate waveforms using Agilent Technologies 3352x waveform generator and output the result in MATLAB®.

Requirements

This example requires the following:

- Keysight IO libraries version 17.1 or newer
- Keysight 335XX / 336XX Function / Arbitrary Waveform Generator IVI driver version 2.2.1.0 or newer

Enumerate Available IVI-C Drivers on the Computer

This enumerates the IVI drivers that have been installed on the computer.

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create MATLAB Instrument Driver And Connect to the Instrument

```
% Create the MATLAB instrument driver
makemid('Ag3352x','Ag3352x.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('Ag3352x.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true');

% Connect driver instance
connect(myInstrument);
```

Attributes and Variables Definition

```
% These values are defined in the driver's header file 'Ag3352x.h'
IVI_ATTR_BASE = 1000000;
IVI_SPECIFIC_ATTR_BASE = IVI_ATTR_BASE + 150000;
```

```
IVI_CLASS_ATTR_BASE = IVI_ATTR_BASE + 250000;
AG3352X_ATTR_CHANNEL_AM_MODE = IVI_SPECIFIC_ATTR_BASE + 46;
AG3352X_ATTR_AM_INTERNAL_DEPTH = IVI_CLASS_ATTR_BASE + 403;
AG3352X_ATTR_AM_INTERNAL_FREQUENCY = IVI_CLASS_ATTR_BASE + 405;
AG3352X_VAL_AM_AMPLITUDE_MODULATION_MODE_ON = 0;
AG3352X_VAL_WFM_SINE = 1;
AG3352X_VAL_AM_INTERNAL = 0;
AG3352X_VAL_AM_INTERNAL_SINE = 1;
```

Get General Device Properties

Query information about the driver and instrument

```
DriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentification');
InstrumentIdentification = get(myInstrument,'Inherentiviattributesinstrumentidentification');
Utility = get(myInstrument, 'Utility');
Revision = invoke(Utility, 'revisionquery');
Vendor = get(DriverIdentification, 'Specific_Driver_Vendor');
Description = get(DriverIdentification, 'Specific_Driver_Description');
InstrumentModel = get(InstrumentIdentification, 'Instrument_Model');
FirmwareRev = get(InstrumentIdentification, 'Instrument_Firmware_Revision');

% Print the queried driver properties
fprintf('Revision:        %s\n', Revision);
fprintf('Vendor:          %s\n', Vendor);
fprintf('Description:     %s\n', Description);
fprintf('InstrumentModel: %s\n', InstrumentModel);
fprintf('FirmwareRev:     %s\n', FirmwareRev);
fprintf(' \n');
```

```
Revision:        2.2.1.0
Vendor:          Agilent Technologies
Description:     IVI driver for the Agilent 33500 family of Function/Arbitrary Waveform Generator
InstrumentModel: 33522B
FirmwareRev:     Sim2.2.1.0
```

Create AM Waveform

```
% Clears all event registers and error queue
DriverStatus = get(myInstrument, 'Driverstatus');
invoke(DriverStatus, 'statusclear');
%Places the instrument in a known state
Utility = get(myInstrument, 'Utility');
invoke(Utility, 'reset');
% Configures standard waveform generation
% AMPLITUDE is set to 5 V,DCOFFSET is set to 0 V,FREQUENCY is set to 1 MHz,
% STARTPHASE is set to 0 degree
ConfigurationStandardWaveform = get(myInstrument, 'Configurationstandardwaveform');
invoke(ConfigurationStandardWaveform, 'configurestandardwaveform', 'Channel1',AG3352X_VAL_WFM_SIN
% Enable amplitude modulation
ConfigurationAmplitudeModulation = get(myInstrument, 'Configurationamplitudemodulation');
invoke(ConfigurationAmplitudeModulation, 'configureamenabled', 'Channel1',true);
% Configures the source of the AM modulating waveform to be internal
invoke(ConfigurationAmplitudeModulation, 'configureamsource', 'Channel1',AG3352X_VAL_AM_INTERNAL)
% Configures internal amplitude modulating waveform source
DEPTH = 50;
AM_FREQUENCY = 10000;
invoke(ConfigurationAmplitudeModulation, 'configureaminternal',DEPTH,AG3352X_VAL_AM_INTERNAL_SIN
```

```
% Enable signal output
Enabled = true;
ConfigurationGeneral = get(myInstrument, 'Configurationgeneral');
invoke(ConfigurationGeneral, 'configureoutputenabled', 'Channel1',Enabled);
```

Display Any Errors from the Driver

```
% If there are any errors, query the driver to retrieve and display them.
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
ErrorQuery: 0, No error.
```

Disconnect Device Object and Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory.
delete(myInstrument);
```

Additional Information:

This example shows the setup and generating of waveforms from a waveform generator using the IVI driver. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Measure Power on a Keysight® RF Power Meter Using the IVI-C Driver

This example shows how to initialize the driver, read a few properties of the driver and make power measurements using Keysight RF Power Meter and output the result in MATLAB®.

Requirements

This example requires the following:

- Keysight (Agilent) IO libraries version 17.1 or newer
- Keysight (Agilent) RF Power Meter IVI version 1.0.9.0 or newer

Enumerate Available IVI-C Drivers On The Computer

This enumerates the IVI drivers that have been installed on the computer.

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create a MATLAB Instrument Driver and Connect to the Simulated Instrument

```
% Create the MATLAB instrument driver
makemid('KtRFPowerMeter','KtRFPowerMeter.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('KtRFPowerMeter.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true')

% Connect driver instance
connect(myInstrument);
```

Attributes and Variables Definition

```
% These values are defined in the driver's header file 'KtRFPowerMeter.h'
IVI_ATTR_BASE = 1000000;
IVI_CLASS_ATTR_BASE = IVI_ATTR_BASE + 250000;
IVI_SPECIFIC_ATTR_BASE = IVI_ATTR_BASE + 150000;
KTRFPOWERMETER_ATTR_CALIBRATOR_ENABLED = IVI_SPECIFIC_ATTR_BASE + 189; % 1150189
KTRFPOWERMETER_ATTR_OFFSET = IVI_CLASS_ATTR_BASE + 5; % 1250005
```

```
KTRFPOWERMETER_ATTR_CHANNELS_ITEM_TRIGGER_CONTINUOUS_ENABLED = IVI_SPECIFIC_ATTR_BASE + 49;
KTRFPOWERMETER_ATTR_MEASUREMENTS_ITEM_OFFSET_ENABLED = IVI_SPECIFIC_ATTR_BASE + 79;
```

Get General Device Properties

Query information about the driver and instrument

```
DriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentification');
InstrumentIdentification = get(myInstrument,'Inherentiviattributesinstrumentidentification');
Utility = get(myInstrument, 'Utility');
Revision = invoke(Utility, 'revisionquery');
Vendor = get(DriverIdentification, 'Specific_Driver_Vendor');
Description = get(DriverIdentification, 'Specific_Driver_Description');
InstrumentModel = get(InstrumentIdentification, 'Instrument_Model');
FirmwareRev = get(InstrumentIdentification, 'Instrument_Firmware_Revision');

% Print the queried driver properties
fprintf('Revision:        %s\n', Revision);
fprintf('Vendor:          %s\n', Vendor);
fprintf('Description:     %s\n', Description);
fprintf('InstrumentModel: %s\n', InstrumentModel);
fprintf('FirmwareRev:     %s\n', FirmwareRev);
fprintf(' \n');
```

```
Revision:        1.0.9.0
Vendor:          Keysight Technologies
Description:     IVI Driver for KtRFPowerMeter [Compiled for 64-bit.]
InstrumentModel: E4416A
FirmwareRev:     Sim1.0.9.0
```

Configure Power Meter

```
% Perform the default preset on the power meter
InstrumentSpecificSystem = get(myInstrument, 'Instrumentspecificsystem');
invoke(InstrumentSpecificSystem, 'systempreset');
% Wait until all instrument operations complete or until MaxTimeMilliseconds has expired
invoke(InstrumentSpecificSystem, 'systemwaitforoperationcomplete', 20000);
% Disables continuous triggering
AttributeAccessors = get(myInstrument, 'Attributeaccessors');
invoke(AttributeAccessors, 'setattributeviboolean', 'A', KTRFPOWERMETER_ATTR_CHANNELS_ITEM_TRIGGE
% Get the number of channels available on the instrument
BasicOperation = get(myInstrument, 'Basicoperation');
ChannelCount = get(BasicOperation, 'Channel_Count');
% Get the model number or name reported by the physical instrument
InherentIviAttributesInstrumentIdentification = get(myInstrument, 'Inherentiviattributesinstrumen
InstrumentModel = get(InherentIviAttributesInstrumentIdentification, 'Instrument_Model');
if (ChannelCount >= 2) && not(strcmpi(InstrumentModel,'N1913A')) && not(strcmpi(InstrumentModel,
    % Disables continuous triggering
    invoke(AttributeAccessors, 'setattributeviboolean', 'B', KTRFPOWERMETER_ATTR_CHANNELS_ITEM_TR
end
% Enables the POWER REF output
invoke(AttributeAccessors, 'setattributeviboolean', 'A', KTRFPOWERMETER_ATTR_CALIBRATOR_ENABLED,
```

Make Measurements

```
% Initiates a measurement on all enabled channels
MeasurementLowLevelMeasurement = get(myInstrument, 'Measurementlowlevelmeasurement');
invoke(MeasurementLowLevelMeasurement, 'initiate');
```

```
% Wait until all instrument operations complete or MaxTimeMilliseconds has expired
invoke(InstrumentSpecificSystem, 'systemwaitforoperationcomplete', 10000);
for iLoop = 1:4
    % Specifying an offset to be added to the measured value in units of dB
    invoke(AttributeAccessors, 'setattributeviréal64', 'A', KTRFPOWERMETER_ATTR_OFFSET, -10*iLoop
    % Initiates a measurement
    InstrumentSpecificMeasurement = get(myInstrument, 'Instrumentspecificmeasurement');
    ReadResult = invoke(InstrumentSpecificMeasurement, 'measurementsitemread', '1',50000);
    % Disable the display offset
    invoke(AttributeAccessors, 'setattributeviboolean', '1', KTRFPOWERMETER_ATTR_MEASUREMENTS_ITE
    % Fetch the result of a previously initiated measurement
    FetchResult = invoke(InstrumentSpecificMeasurement, 'measurementsitemfetch', '1',50000);
    % Enable the display offset
    invoke(AttributeAccessors, 'setattributeviboolean', '1', KTRFPOWERMETER_ATTR_MEASUREMENTS_ITE
    % Perform a power measurement
    MeasResult = invoke(InstrumentSpecificMeasurement, 'measurementsitemmeasure', '1',50000);
    fprintf('Read result: %.3f, Fetch Result: %g, Measure Result: %g\n', ReadResult, FetchResult,
end
fprintf('\n');
```

```
Read result: 0.000, Fetch Result: 0, Measure Result: 0
Read result: 0.000, Fetch Result: 0, Measure Result: 0
Read result: 0.000, Fetch Result: 0, Measure Result: 0
Read result: 0.000, Fetch Result: 0, Measure Result: 0
```

Display Any Errors from the Driver

```
% If there are any errors, query the driver to retrieve and display them.
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
ErrorQuery: 0, No error.
```

Disconnect Device Object and Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory.
delete(myInstrument);
```

Additional Information:

This example shows the setup and measuring of power from a RF power meter using the IVI driver. Once the measured data is retrieved from the instrument, MATLAB can be used to visualize and perform analyses on the data using the rich library of functions in the Signal Processing Toolbox™ and Communications Systems Toolbox™. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Configure Output Signal on Keysight® RF Signal Generator Using the IVI-C Driver

This example shows how to initialize the driver, read a few properties of the driver and configure output signal using Keysight Technologies RF Signal Generators and output the result in MATLAB®.

Requirements

This example requires the following to be installed on the computer:

- Keysight IO libraries version 17.1 or newer
- Keysight RF Signal Generators IVI driver version 1.5.0.0 or newer

Enumerate available IVI-C drivers on the computer

This enumerates the IVI drivers that have been installed on the computer

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create MATLAB Instrument Driver and Connect to the Simulated Instrument

```
% Create the MATLAB instrument driver
makemid('AgRfSigGen','AgRfSigGen.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('AgRfSigGen.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true');

% Connect driver instance
connect(myInstrument);
```

Attributes and Variables Definition

```
% These values are defined in the driver's header file 'AgRfSigGen.h'
AGRFSIGGEN_VAL_LF_GENERATOR_WAVEFORM_SINE = 0;
```

Get General Device Properties

Query information about the driver and instrument

```
DriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentification');
InherentIviAttributesInstrumentIdentification = get(myInstrument, 'Inherentiviattributesinstrumen
Utility = get(myInstrument, 'Utility');
Revision = invoke(Utility, 'revisionquery');
Vendor = get(DriverIdentification, 'Specific_Driver_Vendor');
Description = get(DriverIdentification, 'Specific_Driver_Description');
InstrumentModel = get(InherentIviAttributesInstrumentIdentification, 'Instrument_Model');
FirmwareRev = get(InherentIviAttributesInstrumentIdentification, 'Instrument_Firmware_Revision')

% Print the queried driver properties
fprintf('Revision:        %s\n', Revision);
fprintf('Vendor:          %s\n', Vendor);
fprintf('Description:     %s\n', Description);
fprintf('InstrumentModel: %s\n', InstrumentModel);
fprintf('FirmwareRev:     %s\n', FirmwareRev);
fprintf(' \n');
```

```
Revision:        1.5.0.0
Vendor:          Agilent Technologies
Description:     IVI Driver for Keysight Technologies RF Signal Generator [Compiled for 64-bit.]
InstrumentModel: E4428C
FirmwareRev:     Sim1.5.0.0
```

Output 1GHz/0dBm (1 milliwatt) Carrier Signal

```
fprintf('Carrier Signal:\n\t1GHz/0dBm (1 milliwatt)\n');
invoke(Utility, 'reset');
% Specifies the frequency of the generated RF output signal.
Rf = get(myInstrument,'Rf');
set(Rf, 'Frequency', 1E9); % 1 GHz
% Specifies the amplitude (power/level) of the RF output signal.
set(Rf, 'Power_Level', 0); % 0 dB
% Enable RF output signal.
set(Rf, 'Output_Enabled', true);
```

```
Carrier Signal:
    1GHz/0dBm (1 milliwatt)
```

Disable AM,FM and PM Modes

```
% Disable amplitude modulation (AM) of the RF output signal
AnalogModulationAM = get(myInstrument,'Analogmodulationam');
set(AnalogModulationAM, 'AM_Enabled', false);
% Disable frequency modulation (FM) of the RF output signal
AnalogModulationFM = get(myInstrument,'Analogmodulationfm');
set(AnalogModulationFM, 'FM_Enabled', false);
% Disable phase modulation (PM) of the RF output signal
AnalogModulationPM = get(myInstrument,'Analogmodulationpm');
set(AnalogModulationPM, 'PM_Enabled', false);
```

Enable Amplitude Modulation (AM) Output

```
fprintf('Enable Amplitude Modulation (AM)\n\n');
% Get the analog modulation source name
NameBufferSize = 256;
ConfigurationLfGenerator = get(myInstrument, 'Configurationlfgenerator');
Lfg = invoke(ConfigurationLfGenerator, 'getlfgeneratorname', 1,NameBufferSize);
% Set Lfg as amplitude modulation (AM) source
```

```
set(AnalogModulationAM, 'AM_Source', Lfg);
% Set Lfg as active LF generator
LfGenerator = get(myInstrument, 'Lfgenerator');
set(LfGenerator, 'Active_LFGenerator', Lfg);
% Generate a 2 kHz sine waveform modulation signal
invoke(ConfigurationLfGenerator, 'configurelfgenerator', 2000, AGRFSIGGEN_VAL_LF_GENERATOR_WAVEF
% Enable amplitude modulation (AM) mode
set(AnalogModulationAM, 'AM_Enabled', true);
```

```
Enable Amplitude Modulation (AM)
```

Display Any Errors from the Driver

```
% If there are any errors, query the driver to retrieve and display them
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
ErrorQuery: 0, No error.
```

Disconnect Device Object and Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory
delete(myInstrument);
```

Additional Information:

This example shows the setup and configuring output signal on a RF signal generator using the IVI driver. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Set and Measure DAC (Data Acquisition) Channel Voltage on Keysight® 34970A Switch Using the IVI-C Driver

This example shows how to initialize the driver, read a few properties of the driver, identify installed cards, set DAC channel output voltage and measure DC voltage using Agilent Technologies 34970A data acquisition/switch unit and output the result in MATLAB®.

Requirements

This example requires the following to be installed on the computer:

- Keysight IO libraries version 17.1 or newer
- Keysight 34970A, 34972A Data Acquisition Switch Unit IVI driver version 1.0.3.0 or newer

Enumerate Available IVI-C Drivers On The Computer

This enumerates the IVI drivers that have been installed on the computer.

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create MATLAB Instrument Driver And Connect To The Instrument

```
% Create the MATLAB instrument driver
makemid('Ag34970','Ag34970.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('Ag34970.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true');

% Connect driver instance
connect(myInstrument);
```

Get General Device Properties

Query information about the driver and instrument

```
Utility = get(myInstrument, 'Utility');
DriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentification');
InstrumentIdentification = get(myInstrument,'Inherentiviattributesinstrumentidentification');
```

```
Revision = invoke(Utility, 'revisionquery');
Vendor = get(DriverIdentification, 'Specific_Driver_Vendor');
Description = get(DriverIdentification, 'Specific_Driver_Description');
InstrumentModel = get(InstrumentIdentification, 'Instrument_Model');
FirmwareRev = get(InstrumentIdentification, 'Instrument_Firmware_Revision');

% Print the queried driver properties
fprintf('Revision:        %s\n', Revision);
fprintf('Vendor:          %s\n', Vendor);
fprintf('Description:     %s\n', Description);
fprintf('InstrumentModel: %s\n', InstrumentModel);
fprintf('FirmwareRev:     %s\n', FirmwareRev);
fprintf(' \n');
```

```
Revision:        1.4.0.0
Vendor:          Agilent Technologies
Description:     IVI driver for the Agilent 34970 data acquisition/switch unit. [Compiled for 64-
InstrumentModel: 34970A
FirmwareRev:     Sim1.4.0.0
```

Identify Installed Cards

```
System = get(myInstrument, 'System');
for iLoop = 100:100:300
    CardType = invoke(System, 'systemgetcardtype', iLoop, 127);
    fprintf('Slot: %d  CardType: %s\n', iLoop, CardType);
    if regexp(CardType, ',34907A,')
        DacSlot = iLoop;
    end
    if regexp(CardType, ',34908A,')     % Single ended -- 40 channels
        MuxSlot = iLoop;
    end
    if regexp(CardType, ',34902A,')     % 16 channel Mux
        MuxSlot = iLoop;
    end
    if regexp(CardType, ',34901A,')
        MuxSlot = iLoop;
    end
end
fprintf('\n');
```

```
Slot: 100  CardType: HEWLETT-PACKARD,34901A,0,1.1
Slot: 200  CardType: HEWLETT-PACKARD,34901A,0,1.1
Slot: 300  CardType: HEWLETT-PACKARD,34907A,0,1.1
```

Set and Measure DAC Channel 5 Output Voltage

```
% set DAC voltage to 2.345 V and Select channel 5 on the DAC card
DacVoltage = 2.345;
DacChannel = DacSlot+5;
if DacSlot ~= 0
    % Sets the output voltage level on the DAC channel 5
    Dac = get(myInstrument, 'Dac');
    invoke(Dac, 'dacsetvoltage', DacChannel,DacVoltage);
    % Gets the output voltage level on the DAC channel 5
    DacVoltage = invoke(Dac, 'dacgetvoltage', DacChannel);
    fprintf('DAC channel %d set to %.3f volts\n\n', DacChannel, DacVoltage);
```

```
else
    fprintf('34907A Multifunction Module not found\n\n');
end
```

```
DAC channel 305 set to 2.345 volts
```

Display Errors

```
% If there are any errors, query the driver to retrieve and display them.
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
ErrorQuery: 0, No error.
```

Disconnect Device Object And Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory.
delete(myInstrument);
```

Additional Information:

This example shows the set and measure DAC channel voltage from an Switch using the IVI driver. Once the measured voltage data is retrieved from the instrument, MATLAB can be used to visualize and perform analyses on the data using the rich library of functions in the Signal Processing Toolbox™ and Communications Systems Toolbox™. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

Acquire Signal Spectrum on a Rohde & Schwarz® Spectrum Analyzer Using the IVI-C Driver

This example shows how to initialize the driver, read a few properties of the driver and acquire signal spectrum using Rohde & Schwarz spectrum analyzer and visualize the spectrum in MATLAB®.

Requirements

This example requires the following to be installed on the computer:

- Rohde & Schwarz Spectrum Analyzer IVI-C x64 driver version 1.0 or newer

Enumerate Available IVI-C Drivers On The Computer

This enumerates the IVI drivers that have been installed on the computer.

```
IviInfo = instrhwinfo('ivi');
IviInfo.Modules
```

```
ans =

  Columns 1 through 6

    'Ag33220'    'Ag3352x'    'Ag34410'    'Ag34970'    'Ag532xx'    'AgAC6800'

  Columns 7 through 11

    'AgE36xx'    'AgInfiniiVision'    'AgMD1'    'AgRfSigGen'    'AgXSAn'

  Columns 12 through 13

    'KtRFPowerMeter'    'rsspecan'
```

Create MATLAB Instrument Driver And Connect To The Instrument

```
% Create the MATLAB instrument driver
makemid('rsspecan','rsspecan.mdd')

% Use icdevice with the MATLAB instrument driver name and instrument's
% resource name to create a device object. In this example the instrument
% is connected by GPIB at board index 0 and primary address 1.
myInstrument = icdevice('rsspecan.mdd', 'GPIB0::01::INSTR','optionstring','simulate=true');

% Connect driver instance
connect(myInstrument);
```

Get General Device Properties

Query information about the driver and instrument

```
Utility = get(myInstrument,'Utility');
InherentIviAttributesInstrumentIdentification = get(myInstrument, 'Inherentiviattributesinstrume
InherentIviAttributesDriverIdentification = get(myInstrument,'Inherentiviattributesdriveridentif
Revision = get(InherentIviAttributesDriverIdentification, 'Revision');
Vendor = get(InherentIviAttributesDriverIdentification, 'Driver_Vendor');
```

```
Description = get(InherentIviAttributesDriverIdentification, 'Description');
InstrumentModel = get(InherentIviAttributesInstrumentIdentification, 'Model');
FirmwareRev = get(InherentIviAttributesInstrumentIdentification, 'Firmware_Revision');
% Print the queried driver properties
fprintf('Revision:        %s\n', Revision);
fprintf('Vendor:          %s\n', Vendor);
fprintf('Description:     %s\n', Description);
fprintf('InstrumentModel: %s\n', InstrumentModel);
fprintf('FirmwareRev:     %s\n', FirmwareRev);
fprintf(' \n');
```

```
Revision:        Driver: rsspecan 1.0 (1.0.0.14), Compiler: CVI 13.00, Components: IVIEngine 14.0
Vendor:          Rohde&Schwarz
Description:     Rohde & Schwarz Signal and Spectrum Analyzer IVI-C Driver
InstrumentModel: FSW-26
FirmwareRev:     1.30
```

Attributes and Variables Definition

```
% These constants are defined in the IVI header file 'IviSpecAn.h'
IVISPECAN_VAL_DETECTOR_TYPE_AUTO_PEAK = 1;
IVISPECAN_VAL_VERTICAL_SCALE_LINEAR = 1;
IVISPECAN_VAL_AMPLITUDE_UNITS_DBM = 1;
```

Setting Center frequency and Span

Configure the frequency range of the spectrum analyzer using the center frequency and the frequency span.

```
FrequencyCenter = 2E+9;
FrequencySpan = 500E+6;
Configuration = get(myInstrument, 'Configuration');
invoke(Configuration, 'configurefrequencycenterspan', FrequencyCenter,FrequencySpan);
```

Setting Single Sweep

Configure the acquisition attributes of the spectrum analyzer.

```
SweepModeContinuous = false;
NumberOfSweeps = 1;
DetectorTypeAuto = true;
DetectorType = IVISPECAN_VAL_DETECTOR_TYPE_AUTO_PEAK;
VerticalScale = IVISPECAN_VAL_VERTICAL_SCALE_LINEAR;
Configuration = get(myInstrument, 'Configuration');
invoke(Configuration, 'configureacquisition', SweepModeContinuous,NumberOfSweeps,DetectorTypeAuto
```

Configure Reference Level and Range

Configure the vertical attributes of the spectrum analyzer. This corresponds to attributes like amplitude units, input attenuation, input impedance, reference level, and reference level offset

```
AmplitudeUnits = IVISPECAN_VAL_AMPLITUDE_UNITS_DBM;
InputImpedance = 50.0;
ReferenceLevel = -10.0;
ReferenceLevelOffset = 0;
AttenuationAuto = false;
Attenuation = 10.0;
invoke(Configuration, 'configurelevel', AmplitudeUnits, InputImpedance, ReferenceLevel, Reference
```

Configure Resolution Bandwidth (RBW), Video Bandwidth (VBW) and Sweep Time

Configure the coupling and sweeping attributes of the spectrum analyzer

```
ResolutionBandwidthAuto = false;
ResolutionBandwidth = 1.0E+6;
VideoBandwidthAuto = false;
VideoBandwidth = 1.0E+6;
SweepTimeAuto = false;
SweepTime = 5.0E-3;
invoke(Configuration, 'configuresweepcoupling', ResolutionBandwidthAuto,ResolutionBandwidth,Videc
```

Perform the sweep

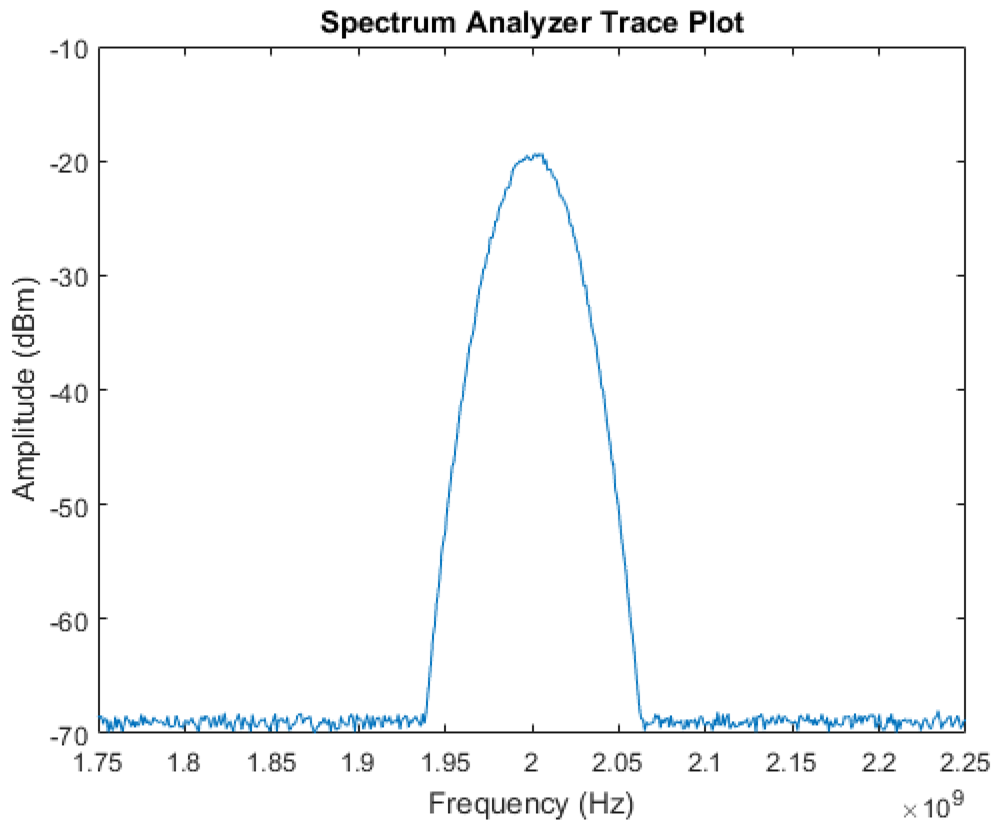
Initiate a signal acquisition based on the present instrument configuration. It then waits for the acquisition to complete, and returns the trace as an array of amplitude values. The amplitude array returns data that represent the amplitude of the signals of the sweep from the start frequency to the stop frequency. The Amplitude Units attribute determines the units of the points in the amplitude array.

```
Tracename = 0;
MaximumTimeMs = 5000;
ArrayLength = 501;
AmplitudeX = 1.75e+09:0.1e+07:2.25e+09;
AmplitudeY = zeros(ArrayLength,1);
Measurement = get(myInstrument, 'Measurement');
[ActualPointsY,AmplitudeY] = invoke(Measurement, 'readytrace', Tracename,MaximumTimeMs,ArrayLeng
```

Visualize Data And Display Any Errors

```
% Display the acquired spectrum
plot(AmplitudeX,AmplitudeY);
title('Spectrum Analyzer Trace Plot');
xlabel('Frequency (Hz)');
ylabel('Amplitude (dBm)');

% If there are any errors, query the driver to retrieve and display them.
ErrorNum = 1;
while (ErrorNum ~= 0)
    [ErrorNum, ErrorMsg] = invoke(Utility, 'errorquery');
    fprintf('ErrorQuery: %d, %s\n', ErrorNum, ErrorMsg);
end
```

```
ErrorQuery: 0, No error.
```



Disconnect Device Object And Clean Up

```
disconnect(myInstrument);
% Remove instrument objects from memory.
delete(myInstrument);
```

Additional Information:

This example shows the setup and acquisition of data from an spectrum analyzer using the IVI driver. Once the measured spectrum is retrieved from the instrument, MATLAB can be used to visualize and perform analyses on the data using the rich library of functions in the Signal Processing Toolbox™ and Communications Systems Toolbox™. Using Instrument Control Toolbox™, it is possible to automate control of instruments, and, build test systems that use MATLAB to perform analyses that may not be possible using the built-in capability of the hardware.

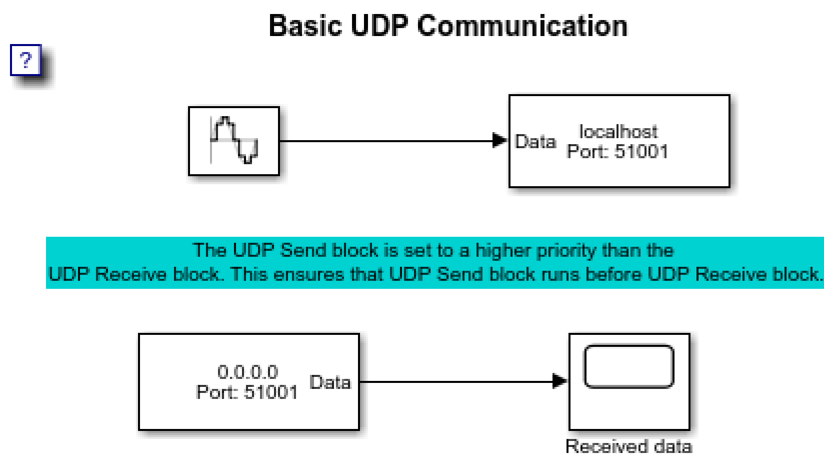
Basic UDP Communication

This example shows how to transfer data over the UDP network using Simulink®.

Instrument Control Toolbox™ provides Simulink® blocks for sending and receiving data over TCP/IP and UDP networks. This example uses the UDP Send and Receive block to perform data transfer over a UDP network.

This example requires Simulink to open and run the model.

```
open_system('demoinstrsl_udpcommunication');
```



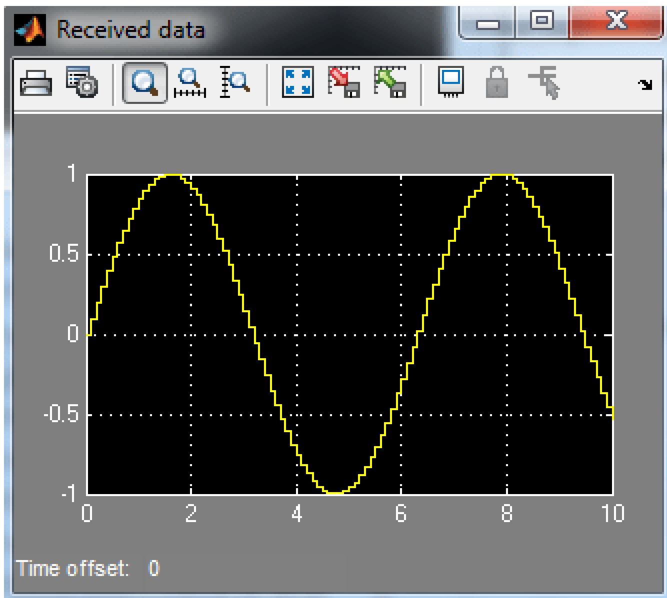

```
close_system('demoinstrsl_udpcommunication');
```

Setup

The input signal sent to the UDP Send block is a sine wave of frequency 1 radians/second. The UDP Send and Receive blocks use the 'localhost' for transferring data across two different ports. The port selected for the two UDP blocks are 51000 and 51001.

Result

The resulting sine wave is seen in the scope block connected to the UDP Receive block.



In this example, the UDP Send and Receive blocks exist in the same Simulink model and are run on the same machine. However, you can also use the UDP Send and Receive blocks in two different models and communicate across two different machines.

Video Surveillance Over TCP/IP Network

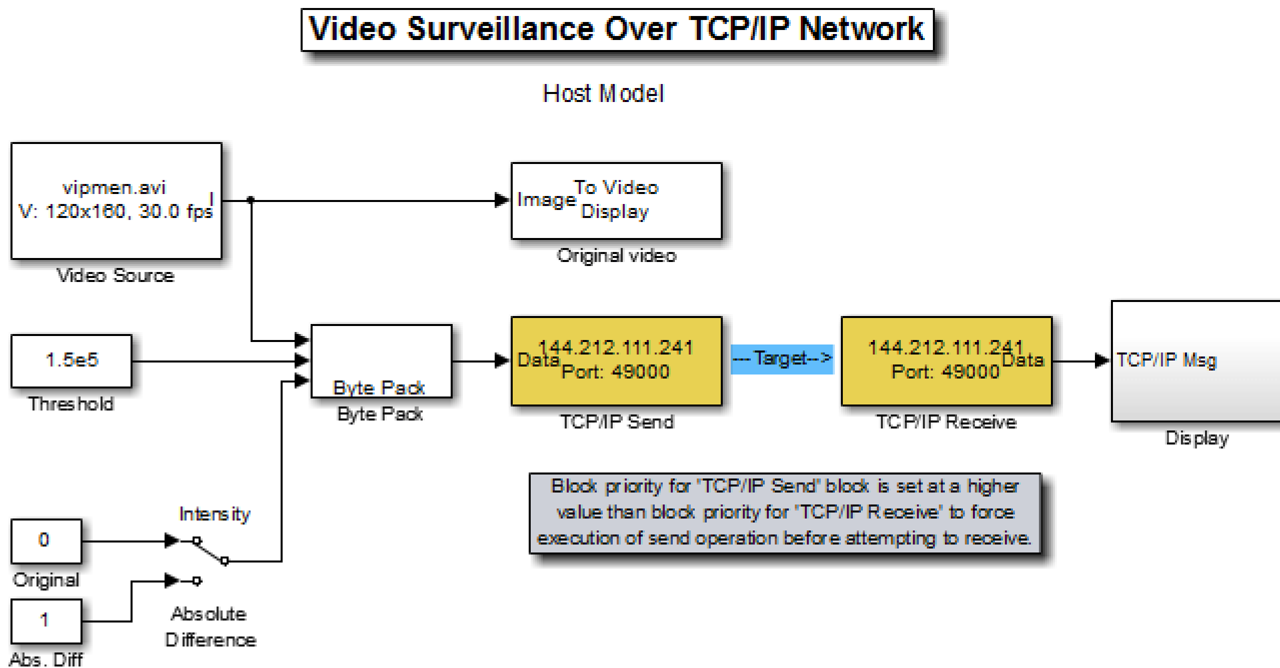
This example shows how to do video surveillance over the TCP/IP network using Simulink®.

Instrument Control Toolbox™ provides Simulink® blocks for sending and receiving data over TCP/IP and UDP networks. This example uses the TCP/IP Send and Receive blocks to perform Video Surveillance. The Simulink model shows a video surveillance recording on your Texas Instruments™ DSP platform using Embedded Coder™ (for TI's C6000™). The motion detection algorithm is implemented in Simulink and deployed to a TIC6000 signal processor.

This example requires Simulink, Computer Vision System Toolbox™, DSP System Toolbox™ and Embedded Coder™ (for TI's C6000™) to open the model. The example also requires the following hardware TMS320C6416 DSK / EVM board, D.signT DSK-91C111 Ethernet daughter card for C6416 DSK target and Ethernet cable.

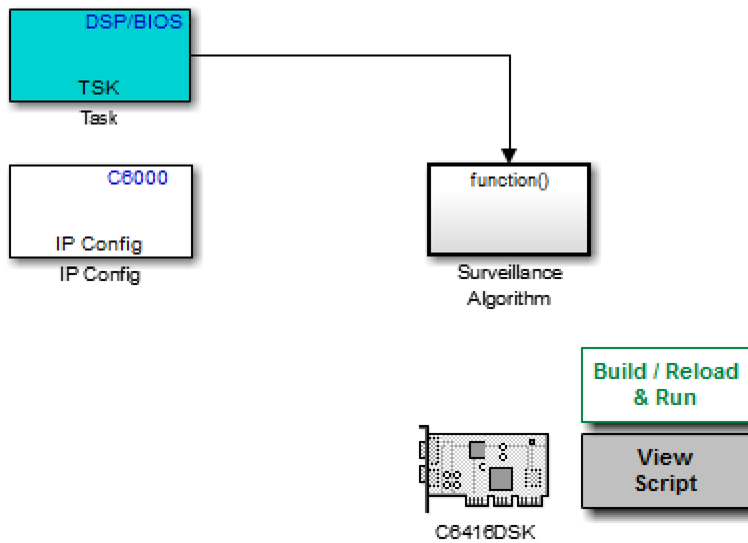
Host Model

The following figure shows the algorithm which runs on the host side machine and communicates with the target using the TCP/IP Send and Receive blocks. The model communicates with the target at port number 49000. The TCP/IP blocks are configured to send and receive data in blocking mode.

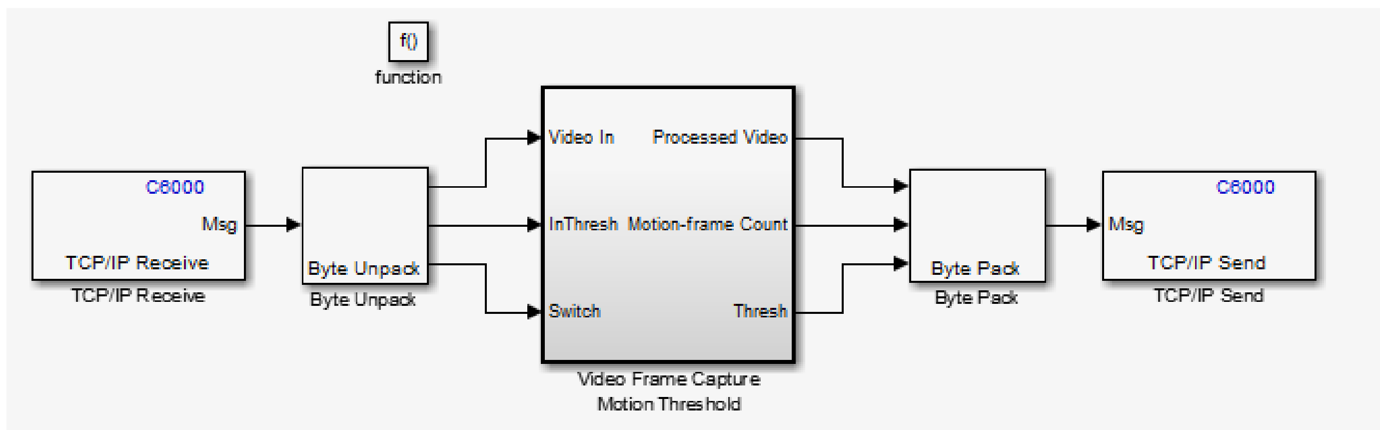


Target Model

The following figure shows the target side example model.



The following figure shows the Surveillance Algorithm which is executed on the target C6416. The algorithm is implemented using blocks from Simulink, DSP System Toolbox™ and Embedded Coder and is converted to C using Simulink® Coder™.



Analysis

While the generated code is running on the target, a host side Simulink model simultaneously sends video frames to the target via TCP/IP protocol. The target receives video frames sent by the host side Simulink model, computes the sum of the absolute value of differences (SAD) between successive video frames, and returns an estimate of motion. When the motion estimate value exceeds a threshold, the target increments a counter and sends the corresponding frame back to the host using the TCP/IP block. You can also adjust the motion threshold using the host side Simulink model.

Running the Example

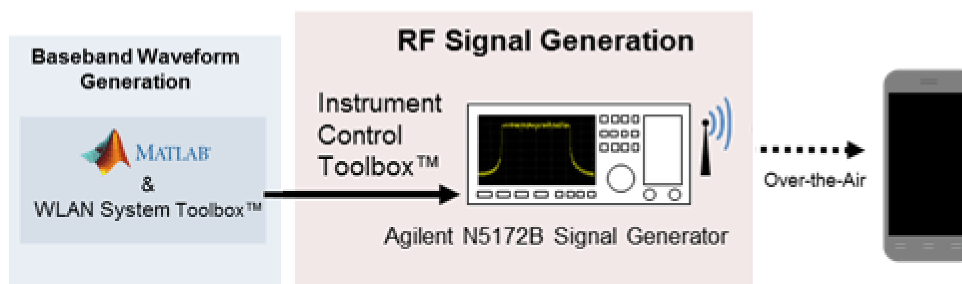
Open the target model and double click "Build Reload & Run" to build, load and run the DSP code. Once the code is generated, it will bring up the host side model. Run the host side model to watch the video surveillance algorithm using motion detection.

802.11 OFDM Beacon Frame Generation and Transmission with Test and Measurement Equipment

This example shows how to generate packets containing MAC beacon frames suitable for baseband simulation or over-the-air transmission using WLAN Toolbox™, Instrument Control Toolbox™ and Keysight Technologies® RF signal generator.

Introduction

In this example WLAN Toolbox is used to create an IEEE® 802.11™ beacon frame. Using Instrument Control Toolbox, the generated beacon frame is downloaded to Keysight Technologies N517B signal generator for over-the-air transmission. Beacon frame is a type of management frame that identifies a basic service set (BSS) formed by a number of 802.11 devices. The access point of a BSS periodically transmits the beacon frame to establish and maintain the network. A WiFi device can be used to view this beacon frame transmitted by the RF Signal Generator.



For more information on beacon frame generation using WLAN Toolbox, refer to “802.11 OFDM Beacon Frame Generation” (WLAN Toolbox).

Requirements

To run this example you need:

- Keysight Technologies N5172B signal generator
- Keysight VISA version 17.3
- IVI-C driver for Keysight Technologies N5172B signal generator
- National Instruments™ IVI® compliance package version 16.0.1.2 or higher
- WLAN Toolbox
- Instrument Control Toolbox

Create IEEE 802.11 Beacon Frame

The beacon packets are periodically transmitted as specified by the Target Beacon Transmission Time (TBTT) in the beacon interval field. The beacon interval represents the number of Time Units (TUs) between TBTT, where 1 TU represents 1024 microseconds. A beacon interval of 100 TU results in a 102.4 milliseconds time interval between successive beacons. A beacon frame is generated using the `wlanMACFrame` (WLAN Toolbox) function. This function consumes the MAC frame configuration object `wlanMACFrameConfig` (WLAN Toolbox). This object accepts `wlanMACManagementConfig` (WLAN Toolbox) as a property to configure the beacon frame-body.

```
SSID = 'TEST_BEACON'; % Network SSID
beaconInterval = 100; % In Time units (TU)
```

```
band = 5;              % Band, 5 or 2.4 GHz
chNum = 52;            % Channel number, corresponds to 5260MHz
bitsPerByte = 8;       % Number of bits in 1 byte

% Create Beacon frame-body configuration object
frameBodyConfig = wlanMACManagementConfig;
frameBodyConfig.BeaconInterval = beaconInterval;  % Beacon Interval in Time units (TUs)
frameBodyConfig.SSID = SSID;                      % SSID (Name of the network)
dsElementID = 3;                                  % DS Parameter IE element ID
dsInformation = dec2hex(chNum, 2);                % DS Parameter IE information
frameBodyConfig = frameBodyConfig.addIE(dsElementID, dsInformation);  % Add DS Parameter IE to th

% Create Beacon frame configuration object
beaconFrameConfig = wlanMACFrameConfig('FrameType', 'Beacon');
beaconFrameConfig.ManagementConfig = frameBodyConfig;

% Generate Beacon frame bits
[mpduBits, mpduLength] = wlanMACFrame(beaconFrameConfig, 'OutputFormat', 'bits');

% Calculate center frequency for the given band and channel number
fc = helperWLANChannelFrequency(chNum, band);
```

Create IEEE 802.11 Beacon Packet

A beacon packet is synthesized using `wlanWaveformGenerator` (WLAN Toolbox) with a non-HT format configuration object. In this example an object is configured to generate a beacon packet of 20 MHz bandwidth, 1 transmit antenna and BPSK rate 1/2 (MCS 0).

```
cfgNonHT = wlanNonHTConfig;              % Create a wlanNonHTConfig object
cfgNonHT.PSDULength = numel(mpduBits)/8; % Set the PSDU length in bits

% The idle time is the length in seconds of an idle period after each
% generated packet. The idle time is set to the beacon interval.
txWaveform = wlanWaveformGenerator(mpduBits, cfgNonHT, 'IdleTime', beaconInterval*1024e-6);
Rs = wlanSampleRate(cfgNonHT);           % Get the input sampling rate
```

Create a RF Signal Generator Object

Quick-Control RF Signal Generator is used to download and transmit the baseband waveform, `txWaveform`, generated by WLAN Toolbox.

```
rf = rfsiggen();
```

Discover all the available instrument resources you can connect to, using the `resources` method.

```
rf.resources
```

```
ans =

    ' ASRL::COM3
      PXI0::MEMACC
      TCPIP0::172.31.165.249::inst0::INSTR
     '
```

Discover all the available instrument drivers, using `drivers` method.

```
rf.drivers
```

```
ans =

    'Driver: AgRfSigGen_SCPI
     Supported Models:
     E4428C, E4438C

     Driver: RsRfSigGen_SCPI
     Supported Models:
     SMW200A, SMBV100A, SMU200A, SMJ100A, AMU200A, SMATE200A

     Driver: AgRfSigGen
     Supported Models:
        E4428C,E4438C,N5181A,N5182A,N5183A,N5171B,N5181B,N5172B
        N5182B,N5173B,N5183B,N5166B,N5182N,E8241A,E8244A,E8251A
        E8254A,E8247C,E8257C,E8267C,E8257D,E8267D,E8663B,E8257N

     Driver: nisRFSigGen
     Supported Models:


     '
```

Connect to Signal Generator

Set `Resource` and `Driver` properties before connecting to the object. The IP address of Keysight Technologies N5172B signal generator is *172.31.165.249* , hence the resource specified will be 'TCPIP0::172.31.165.249::inst0::INSTR'

```
rf.Resource = 'TCPIP0::172.31.165.249::inst0::INSTR';
rf.Driver = 'AgRfSigGen';
% Connect to the instrument
connect(rf);
```

Download Waveform

Download the waveform, `txWaveform`, to the instrument with sampling rate `Rs`.

```
download(rf, transpose(txWaveform), Rs);
```

Transmit the Waveform

Call `start` to start transmitting waveform using specified centerFrequency, outputPower and loopCount.

```
centerFrequency = fc;
outputPower = 0;
loopCount = Inf;
start(rf, centerFrequency, outputPower, loopCount);
```

Once the signal generator is transmitting the beacon, you can test by scanning for wireless network using a Wi-Fi device. You should now see a *TEST_BEACON* SSID in the list of available networks.

Clean Up

When you have finished transmitting, stop the waveform output, disconnect the `rfsiggen` object from the signal generator, and remove it from the workspace.

```
stop(rf);
disconnect(rf);
clear rf
```

Appendix

This example uses the following helper functions:

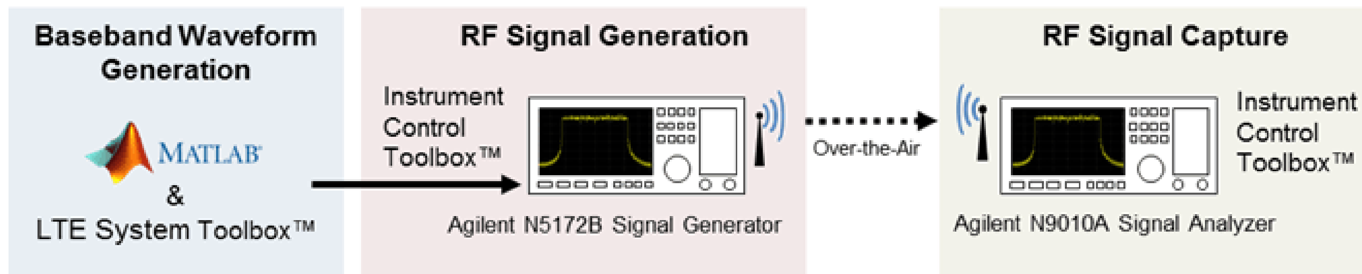
- helperWLANChannelFrequency.m

LTE Waveform Generation and Transmission Using Quick Control RF Signal Generator

This example shows how an over-the-air LTE waveform can be generated and transmitted using LTE Toolbox™, Instrument Control Toolbox™ and Keysight Technologies® RF instruments.

Introduction

In this example LTE Toolbox is used to generate a standard baseband IQ downlink test model (E-TM) waveform. Using Instrument Control Toolbox, the generated waveform is downloaded to Keysight Technologies N5172B signal generator for over-the-air transmission. The over-the-air signal is captured using an Keysight Technologies N9010A signal analyzer.



For more information on LTE waveform generation and analysis, refer to "Waveform Generation and Transmission using LTE Toolbox with Test and Measurement Equipment" (LTE Toolbox).

Requirements

To run this example you need:

- Keysight Technologies N5172B signal generator
- Keysight Technologies N9010A signal analyzer
- Keysight VISA version 17.3
- IVI-C driver for Keysight Technologies N5172B signal generator
- National Instruments™ IVI® compliance package version 16.0.1.2 or higher
- LTE Toolbox
- Instrument Control Toolbox

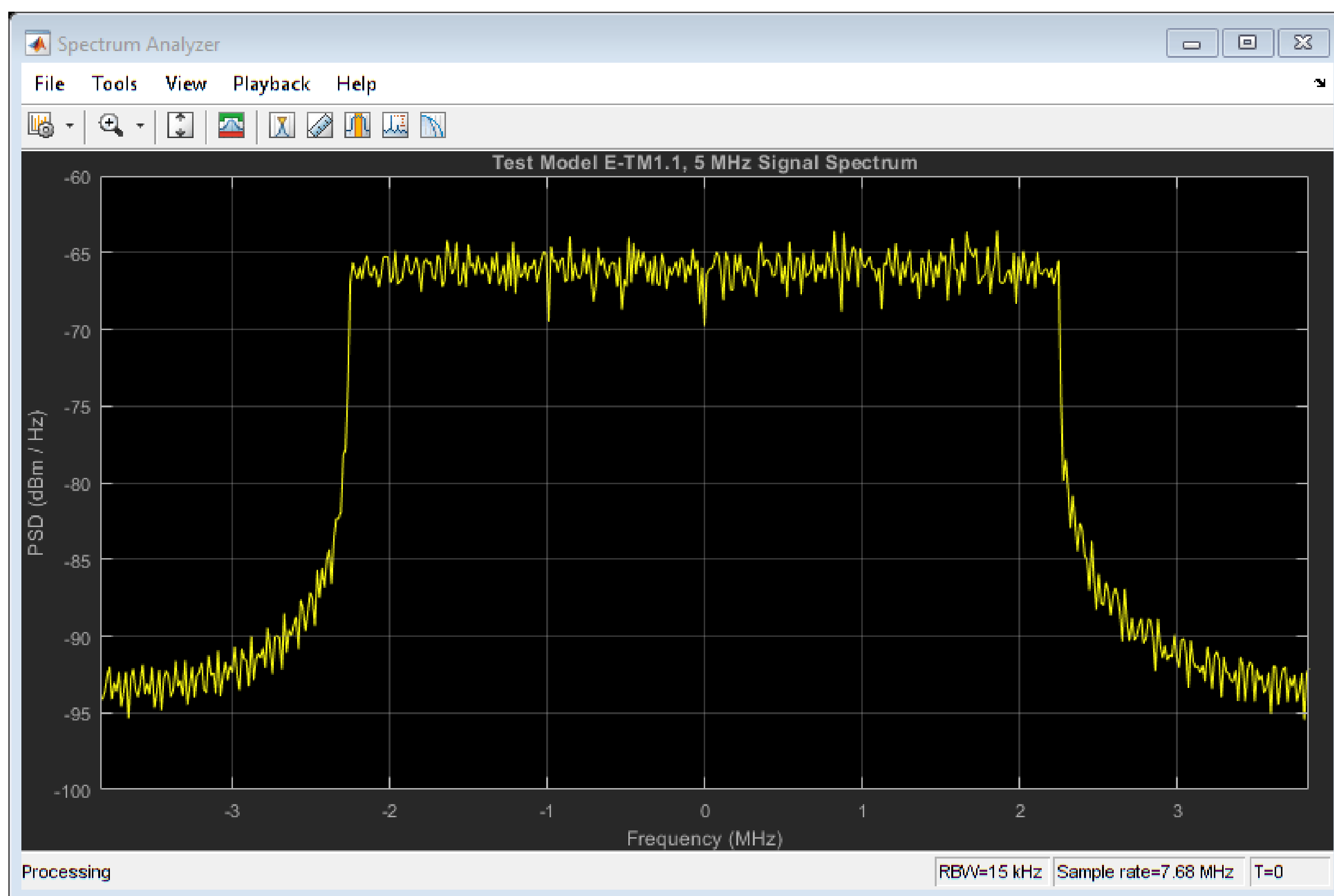
Generate a Baseband Waveform Using the LTE Toolbox

Generate test model waveform using `lteTestModelTool` (LTE Toolbox), this returns an E-TM time-domain waveform, `waveform`, a 2-D numeric array of resource elements for a number of subframes across a single antenna port, `tmgrid`, and a scalar structure, `tmconfig`, containing information about the OFDM modulated waveform.

```
config = lteTestModel('1.1', '5MHz');  % Test Model number 1.1, 5MHz bandwidth
config.TotSubframes = 100;             % Generate 100 subframes
[waveform, tmgrid, tmconfig] = lteTestModelTool(config);
```

The frequency spectrum of the generated time domain waveform, `waveform`, can be viewed using the `dsp.SpectrumAnalyzer` (DSP System Toolbox). As expected, the 5MHz signal bandwidth is clearly visible at baseband.

```
% Calculate the spectral content in the LTE signal
spectrumPlotTx = dsp.SpectrumAnalyzer;
spectrumPlotTx.SampleRate = tmconfig.SamplingRate;
spectrumPlotTx.SpectrumType = 'Power density';
spectrumPlotTx.PowerUnits =  'dBm';
spectrumPlotTx.RBWSource = 'Property';
spectrumPlotTx.RBW = 15e3;
spectrumPlotTx.FrequencySpan = 'Span and center frequency';
spectrumPlotTx.Span = 7.68e6;
spectrumPlotTx.CenterFrequency = 0;
spectrumPlotTx.Window = 'Rectangular';
spectrumPlotTx.SpectralAverages = 10;
spectrumPlotTx.YLimits = [-100 -60];
spectrumPlotTx.YLabel = 'PSD';
spectrumPlotTx.Title = 'Test Model E-TM1.1, 5 MHz Signal Spectrum';
spectrumPlotTx.ShowLegend = false;
spectrumPlotTx(waveform);
```



Generate an Over-the-Air Signal using Quick-Control RF Signal Generator

Quick-Control RF Signal Generator is used to download and transmit the test model waveform created by the LTE Toolbox, `waveform`, using the Agilent Technologies N5172B signal generator. This creates an RF LTE signal with a center frequency of 1GHz. Note 1GHz was selected as an example frequency and is not intended to be a recognized LTE channel.

Create an RF Signal Generator object

```
rf = rfsiggen();
```

Discover all the available instrument resources you can connect to, using the `resources` method.

```
rf.resources
```

```
ans =

    ' ASRL1::INSTR
      ASRL3::INSTR
      ASRL::COM1
      ASRL::COM3
      TCPIP0::172.28.21.217::inst0::INSTR
     '
```

Discover all the available instrument drivers, using `drivers` method.

```
rf.drivers
```

```
ans =

    'Driver: AgRfSigGen_SCPI
     Supported Models:
     E4428C, E4438C

     Driver: RsRfSigGen_SCPI
     Supported Models:
     SMW200A, SMBV100A, SMU200A, SMJ100A, AMU200A, SMATE200A

     Driver: AgRfSigGen
     Supported Models:
        E4428C,E4438C,N5181A,N5182A,N5183A,N5171B,N5181B,N5172B
        N5182B,N5173B,N5183B,E8241A,E8244A,E8251A,E8254A,E8247C'
```

Set `Resource` and `Driver` properties before connecting to the object. The IP address of Keysight Technologies N5172B signal generator is *172.28.21.217* , hence the resource specified will be 'TCPIP0::172.28.21.217::inst0::INSTR'

```
rf.Resource = 'TCPIP0::172.28.21.217::inst0::INSTR';
rf.Driver = 'AgRfSigGen';
% Connect to the instrument
connect(rf);
```

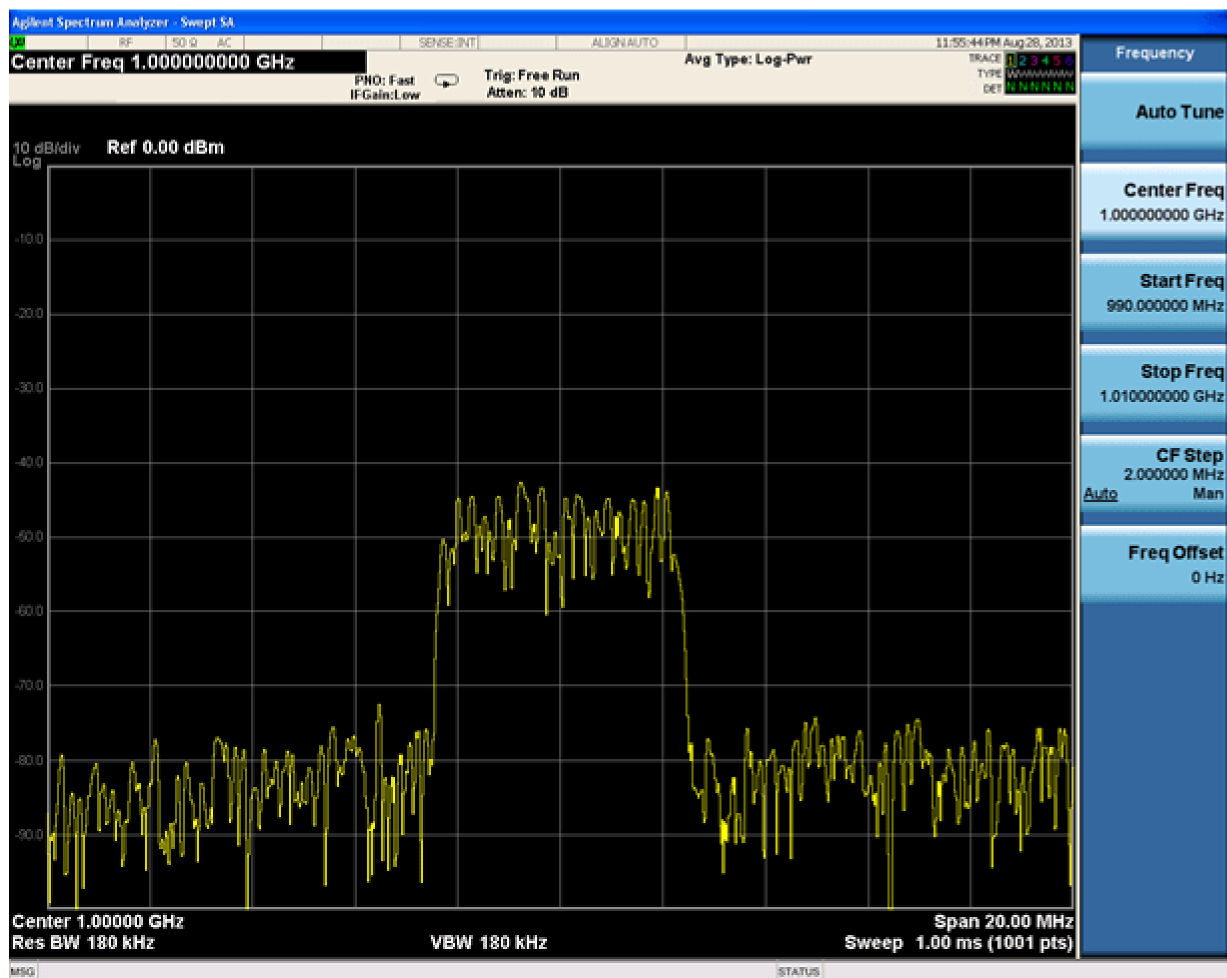
Download the waveform, `waveform`, to the instrument

```
download(rf, transpose(waveform), tmconfig.SamplingRate);
```

Call `start` to start transmitting waveform using specified `centerFrequency`, `outputPower` and `loopCount`. Loop count represents the number of times the waveform should be repeated.

```
centerFrequency = 1e9;
outputPower = 0;
loopCount = Inf;
start(rf, centerFrequency, outputPower, loopCount);
```

The frequency spectrum of the RF signal transmitted by the signal generator can be viewed using a spectrum analyzer tuned to the 1GHz center frequency. The screen capture below, from an Agilent Technologies N9010A signal analyzer, clearly shows the 5MHz signal bandwidth.



Clean up

When you have finished transmitting data, stop the waveform output, disconnect the `rfsiggen` object from the signal generator, and remove it from the workspace.

```
stop(rf);
disconnect(rf);
clear rf
```

Creating and Downloading an IQ Waveform to a RF Signal Generator

This example shows how to use the Quick-Control RF Signal Generator to generate and transmit RF waveforms.

Introduction

In this example we will create an IQ waveform and transmit this waveform using the Quick-Control RF Signal Generator.

Requirements

To run this example you need:

- Keysight Technologies® N5172B signal generator
- Keysight VISA version 17.3
- IVI-C driver for Keysight Technologies N5172B signal generator
- National Instruments™ IVI® compliance package version 16.0.1.2 or higher

Create IQ waveform

We will create an IQ waveform that consists of two sinusoid signals with real and imaginary values.

When generating signals for the RF Signal Generator ensure that the waveform is a continuous row vector.

```
% Configure parameters for waveform.

% Number of points in the waveform
points = 1000;

% Determines the frequency offset from the carrier
cycles = 101;
phaseInc = 2*pi*cycles/points;
phase = phaseInc * (0:points-1);

% Create an IQ waveform
Iwave = cos(phase);
Qwave = sin(phase);
IQData = Iwave+1i*Qwave;
IQData = IQData(:)';
```

Create an RF Signal Generator Object

```
rf = rfsiggen();
```

Discover all the available instrument resources you can connect to, using the `resources` method.

```
rf.resources
```

```
ans =

    ' ASRL1::INSTR
```

```
    ASRL3::INSTR
    ASRL::COM1
    ASRL::COM3
    PXI0::MEMACC
    TCPIP0::172.28.22.99::inst0::INSTR
    TCPIP0::A-N5172B-50283.dhcp.mathworks.com::inst0::INSTR
    TCPIP0::A-N9010A-21026.dhcp.mathworks.com::inst0::INSTR
   '
```

Discover all the available instrument drivers, using `drivers` method.

```
rf.drivers
```

```
ans =

    'Driver: AgRfSigGen_SCPI
     Supported Models:
     E4428C, E4438C

     Driver: RsRfSigGen_SCPI
     Supported Models:
     SMW200A, SMBV100A, SMU200A, SMJ100A, AMU200A, SMATE200A

     Driver: AgRfSigGen
     Supported Models:
        E4428C,E4438C,N5181A,N5182A,N5183A,N5171B,N5181B,N5172B
        N5182B,N5173B,N5183B,E8241A,E8244A,E8251A,E8254A,E8247C

     Driver: nisRFSigGen
     Supported Models:'
```

Connect to Signal Generator

Set `Resource` and `Driver` property before connecting to the object.

```
rf.Resource = 'TCPIP0::A-N5172B-50283.dhcp.mathworks.com::inst0::INSTR';
rf.Driver = 'AgRfSigGen';
% Connect to the instrument
connect(rf);
```

Download the Waveform

Download the waveform, `IQData` to the instrument with sampling rate of 10MHz.

```
samplingRate = 10e6;
download(rf, IQData, samplingRate);
```

Transmit the Waveform

Transmit the downloaded waveform with center frequency of 1GHz and output power of 0dBm. Note these values are selected as reference values and is not intended to be recognized as standard values for transmitting any RF signals. Loop count represents the number of times the waveform should be repeated.

```
centerFrequency = 1e9;
outputPower = 0;
```

```
loopCount = Inf;
start(rf, centerFrequency, outputPower, loopCount);
```

Stop Transmitting the Waveform

Once you have finished transmitting the signal, stop the transmission.

```
stop(rf);
```

Clean Up

Close the connection of the signal generator and remove it from the workspace.

```
disconnect(rf);
delete(rf);
clear rf
```

Fetch Spectrum Through Ocean Optics Spectrometer Using MATLAB Instrument Driver

This example shows how to acquire the spectrum of a fluorescent light source from an Ocean Optics Spectrometer.

Introduction

Instrument Control Toolbox™ supports communication with instruments through high-level drivers. In this example you can acquire spectrum from an Ocean Optics Spectrometer using the MATLAB Instrument Driver.

Requirements

This example requires the following:

- A 64-bit Microsoft® Windows®
- Ocean Optics spectrometer USB2000

Create MATLAB Instrument OmniDriver object.

```
spectrometerObj = icdevice('OceanOptics_OmniDriver.mdd');
```

Connect to the instrument.

```
connect(spectrometerObj);
disp(spectrometerObj)
```

```
NatUSB_64
Driver: NatUSBWin_64

   Instrument Device Object Using Driver : OceanOptics_OmniDriver.mdd

   Instrument Information
      Type:               Spectrometer
      Manufacturer:       Ocean Optics
      Model:              QE65 Pro, Maya2000 Pro, Jaz EL350, HR2000, USB2000, USB4000, NIRQuest

   Driver Information
      DriverType:         MATLAB generic
      DriverName:         OceanOptics_OmniDriver.mdd
      DriverVersion:      1.0

   Communication State
      Status:             open
```

Set parameters for spectrum acquisition.

```
% integration time for sensor.
integrationTime = 50000;
% Spectrometer index to use (first spectrometer by default).
spectrometerIndex = 0;
% Channel index to use (first channel by default).
channelIndex = 0;
% Enable flag.
enable = 1;
```

Identify the spectrometer connected.

```
% Get number of spectrometers connected.
numOfSpectrometers = invoke(spectrometerObj, 'getNumberOfSpectrometersFound');

disp(['Found ' num2str(numOfSpectrometers) ' Ocean Optics spectrometer(s).'])

% Get spectrometer name.
spectrometerName = invoke(spectrometerObj, 'getName', spectrometerIndex);
% Get spectrometer serial number.
spectrometerSerialNumber = invoke(spectrometerObj, 'getSerialNumber', spectrometerIndex);
disp(['Model Name : ' spectrometerName])
disp(['Model S/N  : ' spectrometerSerialNumber])
```

```
Found 1 Ocean Optics spectrometer(s).
Model Name : USB2000+
Model S/N  : USB2+H11505
```

Set the parameters for spectrum acquisition.

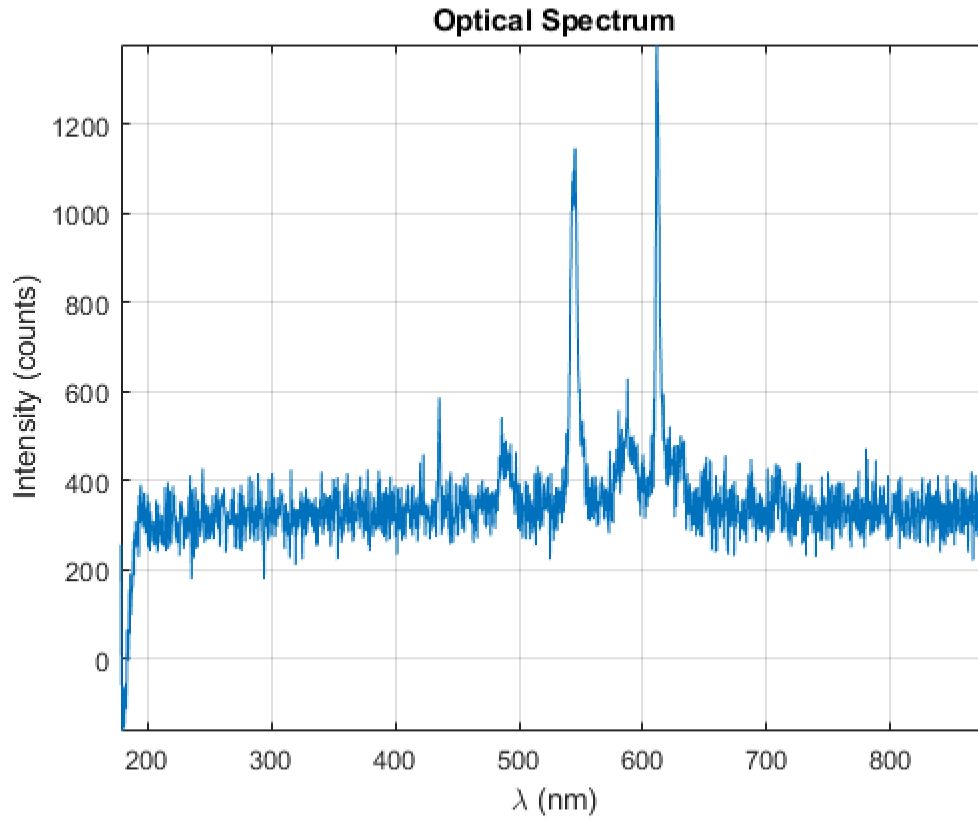
```
% Set integration time.
invoke(spectrometerObj, 'setIntegrationTime', spectrometerIndex, channelIndex, integrationTime);
% Enable correct for detector non-linearity.
invoke(spectrometerObj, 'setCorrectForDetectorNonlinearity', spectrometerIndex, channelIndex, ena
% Enable correct for electrical dark.
invoke(spectrometerObj, 'setCorrectForElectricalDark', spectrometerIndex, channelIndex, enable);
```

Acquire the spectrum.

```
wavelengths = invoke(spectrometerObj, 'getWavelengths', spectrometerIndex, channelIndex);
% Get the wavelengths of the first spectrometer and save them in a double
% array.
spectralData = invoke(spectrometerObj, 'getSpectrum', spectrometerIndex);
```

Plot the waveform.

```
    plot(wavelengths, spectralData);
    title('Optical Spectrum');
    ylabel('Intensity (counts)');
    xlabel('\lambda (nm)');
    grid on
    axis tight
```



Clean up.

```
disconnect(spectrometerObj);
```

```
delete (spectrometerObj);
```

Read Streaming Data from Arduino Using Serial Port Communication

This example shows how to enable callbacks to read streaming ASCII terminated data from an Arduino® Due using the `serialport` interface.

Load Program on the Arduino

Plug in an Arduino Due to your computer.

Load the following program on the Arduino Due using the Arduino IDE. This program writes out continuous points of a sine wave, followed by the "Carriage Return" and "Linefeed" terminators.

```
/*
 SineWavePoints

 Write sine wave points to the serial port, followed by the Carriage Return and LineFeed terminat
 */

int i = 0;

// The setup routine runs once when you press reset:
void setup() {
  // Initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
}

// The loop routine runs over and over again forever:
void loop() {
  // Write the sinewave points, followed by the terminator "Carriage Return" and "Linefeed".
  Serial.print(sin(i*50.0/360.0));
  Serial.write(13);
  Serial.write(10);
  i += 1;
}
```

Establish a Connection to the Arduino

Create a `serialport` instance to connect to your Arduino Due.

Find the serial port that the Arduino is connected to. You can identify the port from the Arduino IDE.

```
serialportlist("available")'
```

```
ans = 3×1 string
    "COM1"
    "COM3"
    "COM13"
```

Connect to the Arduino Due by creating a `serialport` object using the port and baud rate specified in the Arduino code.

```
arduinoObj = serialport("COM13",9600)
```

```
arduinoObj =
Serialport with properties
```

```
                 Port: "COM13"
             BaudRate: 9600
    NumBytesAvailable: 0
      NumBytesWritten: 0

Show all properties
```

Prepare the `serialport` Object to Start Streaming Data

Configure the `serialport` object by clearing old data and configuring its properties.

Set the `Terminator` property to match the terminator that you specified in the Arduino code.

```
configureTerminator(arduinoObj,"CR/LF");
```

Flush the `serialport` object to remove any old data.

```
flush(arduinoObj);
```

Prepare the `UserData` property to store the Arduino data. The `Data` field of the struct saves the sine wave value and the `Count` field saves the x-axis value of the sine wave.

```
arduinoObj.UserData = struct("Data",[],"Count",1)
```

```
arduinoObj = 
Serialport with properties

                 Port: "COM13"
             BaudRate: 9600
    NumBytesAvailable: 10626
      NumBytesWritten: 0

Show all properties
```

Create a callback function `readSineWaveData` that reads the first 1000 ASCII terminated sine wave data points and plots the result.

```
function readSineWaveData(src, ~)

% Read the ASCII data from the serialport object.
data = readline(src);

% Convert the string data to numeric type and save it in the UserData
% property of the serialport object.
src.UserData.Data(end+1) = str2double(data);

% Update the Count value of the serialport object.
src.UserData.Count = src.UserData.Count + 1;

% If 1001 data points have been collected from the Arduino, switch off the
% callbacks and plot the data.
if src.UserData.Count > 1001
    configureCallback(src, "off");
    plot(src.UserData.Data(2:end));
end
end
```

Set the `BytesAvailableFcnMode` property to "`terminator`" and the `BytesAvailableFcn` property to `@readSineWaveData`. The callback function `readSineWaveData` is triggered when a new sine wave data (with the terminator) is available to be read from the Arduino.

```
configureCallback(arduinoObj,"terminator",@readSineWaveData);
```

The callback function opens the MATLAB figure window with a plot of the first 1000 sine wave data points.

Read Waveform from Tektronix TDS 1002 Scope Using SCPI Commands

This example shows how to configure a Tektronix TDS 1002 scope and read a waveform from the scope using the scope specific SCPI commands.

Connect to the Scope

Connect a Tektronix TDS 1002 scope to your computer. On the scope, press the Utility button, then select Options followed by RS232 Setup. Set the following configuration:

- EOL string to "CR/LF"
- Baud to 9600
- Flow control to None
- Parity to None

Connect to the scope using the `serialport` function. Specify the port that your scope is connected to on your computer. In this example, the port is `"COM1"`. Set the `BaudRate` to 9600 to match what you set on the scope.

```
s = serialport("COM1",9600)
```

```
s = 
  Serialport with properties:

                 Port: "COM1"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, all methods
```

Configure the Terminator

Set the `Terminator` property for the `serialport` object using `configureTerminator`. The `Terminator` property matches the EOL string of the scope.

```
configureTerminator(s,"CR/LF")
terminator = s.Terminator
```

```
terminator = 1×1 string
"CR/LF"
```

Query the scope with the new `Terminator` value using the SCPI Command `"*IDN?"` in `writeread`. If your scope is connected and `Terminator` is configured properly, then this returns a string that uniquely identifies the scope.

```
scopeID = writeread(s,"*IDN?")
```

```
scopeID = 1×1 string
"TEKTRONIX,TDS 1002,0,CF:91.1CT FV:v2.12 TDS2CM:CMV:v1.04"
```

Configure the Channel

Configure Channel 1 of the scope using `writeline`. Write the SCPI commands as ASCII terminated string data to the `serialport` object. Then, confirm that Channel 1 is set as the source using `writeread`.

```
writeline(s,"HEADER OFF")
writeline(s,"DATA:SOURCE CH1")
scopeSource = writeread(s,"DATA:SOURCE?")
```

```
scopeSource = 1×1 string
"CH1"
```

Set the waveform data encoding method to Most Significant Bit (MSB) transferred first, using `writeline`. Confirm the encoding method using `writeread`.

```
writeline(s,"DATA:ENCDG RIBINARY");
scopeEncodingMethod = writeread(s,"DATA:ENCDG?")
```

```
scopeEncodingMethod = 1×1 string
"RIB"
```

Get Waveform Information

Get the waveform transmission and formatting settings.

```
scopeWaveformPreamble = writeread(s,"WFMpre?")
```

```
scopeWaveformPreamble = 1×1 string
"1;8;BIN;RI;MSB;2500;"Ch1, DC coupling, 2.0E0 V/div, 5.0E-4 s/div, 2500 points, Sample mode";Y;2
```

Get the number of points in the waveform.

```
scopeNumPoints = writeread(s,"WFMpre:NR_Pt?")
```

```
scopeNumPoints = 1×1 string
"2500"
```

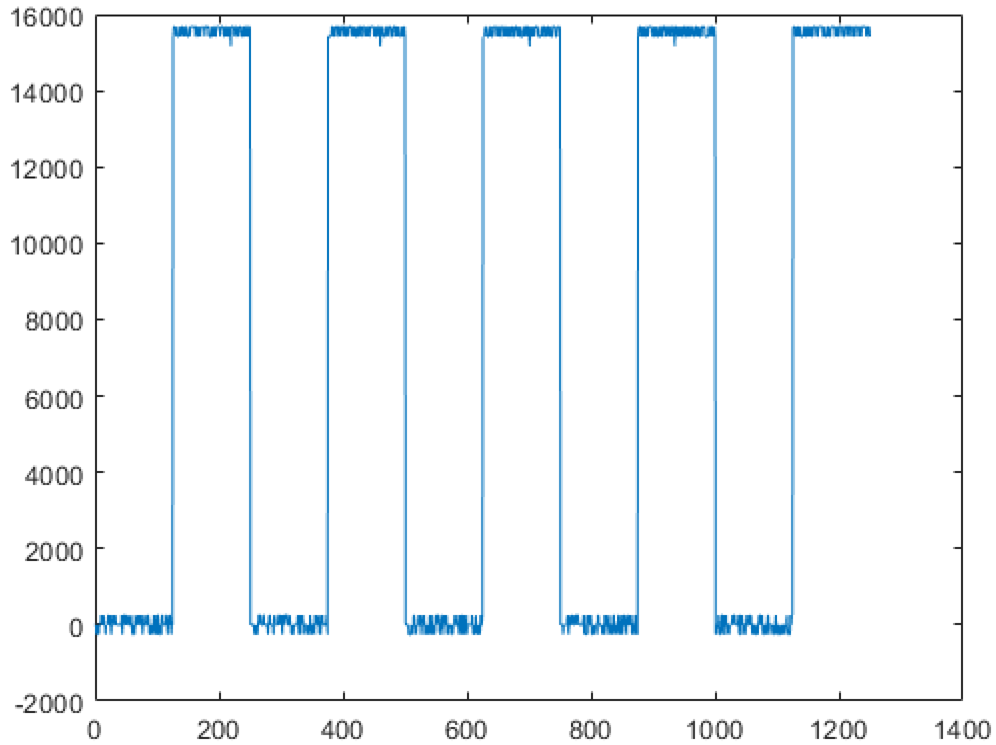
Read and Plot the Waveform

Get the waveform data from the scope to MATLAB using the `"CURVE?"` command and read the waveform back into MATLAB using `readbinblock`.

```
writeline(s,"CURVE?")
scopeWaveform = readbinblock(s,"int16");
```

Plot the waveform.

```
plot(scopeWaveform)
```



Clear the Connection

When you are finished working with the scope, clear the `serialport` object.

```
clear s
```

Read Voltage Through NI-DMM MATLAB Instrument Driver in Simulation Mode

This example shows how to read voltage from a National Instruments® NI-DMM driver in the simulation mode.

Requirements

This example requires a Microsoft® Windows® system and NI-DMM package 3.06 or higher. Make sure the Measurement & Automation Explorer recognizes the NI-DMM device before you use this example.

Verify NI-DMM Installation

Use the `instrhwinfo` command to check if the NI-DMM software package is installed correctly. If installed correctly, NI-DMM is listed as one of the modules installed on the Windows machine. This example uses libraries installed with it.

```
driversInfo = instrhwinfo ('ivi');
disp(driversInfo.Modules');
```

```
    {'nidcpower'       }
    {'nidmm'           }
    {'niFgen'          }
    {'nisACPwr'        }
    {'niScope'         }
    {'nisCounter'      }
    {'nisDCPwr'        }
    {'nisDigitizer'    }
    {'nisDmm'          }
    {'nisDownconverter'}
    {'nisFGen'         }
    {'nisPwrMeter'     }
    {'nisRFSigGen'     }
    {'nisScope'        }
    {'nisSpecAn'       }
    {'nisSwtch'        }
    {'nisUpconverter'  }
    {'niSwitch'        }
```

Create a MATLAB Instrument Object

Use the `icdevice` function to create an instrument object from the MDD that was part of the NI-DMM support package, and establish a connection to the DMM using that object.

The `icdevice` function takes two or more input arguments: the MDD file name, the resource name for the DMM, and optional device-specific parameters.

You can get the resource name for the DMM from NI Measurement and Automation Explorer. For example: A resource name of `PXI1Slot6` in NI MAX would be `DAQ::PXI1Slot6` and `Device 1` would be `DAQ::Dev1`. You can remove the `optionstring` argument and the corresponding string parameter if you have the actual hardware.

You can establish a connection to the DMM using the `connect` command.

```
ictObj = icdevice('nidmm.mdd', 'DAQ::Dev1', 'optionstring','simulate=true');
connect(ictObj);
disp(ictObj);
```

```
   Instrument Device Object Using Driver : niDMM

   Instrument Information
      Type:               IVIInstrument
      Manufacturer:       National Instruments Corp.
      Model:              National Instruments Digital Multimeters

   Driver Information
      DriverType:         MATLAB IVI
      DriverName:         niDMM
      DriverVersion:      1.0

   Communication State
      Status:             open
```

Configure the DMM

For the purpose of this example, the DMM is configured as

```
* Measurement Function: DC Voltage

* Range: 10V

* Resolution: 5.5 Digits
```

Use the MATLAB Instrument Driver Editor `midedit` to view other properties and functions that allow you to configure a device. The tool shows all the properties and functions that the NI-DMM software package supports.

The Measurement Function value for DC Voltage is 1. Measurement Function will have different values for other measurement types such as AC Voltage, DC Current, etc.

```
measurementFunction = 1;
range = 10;
resolution = 5.5;

configuration = get(ictObj, 'configuration');
invoke(configuration, 'configuremeasurementdigits', measurementFunction, range, resolution);
```

Read and Display the Voltage

Once you configure the DMM with the required settings, use an appropriate function call to read the voltage.

```
% Configure DMM to calculate the timeout automatically
AutoTimeLimit = -1;

acquisition = get(ictObj, 'acquisition');
volts =  invoke(acquisition, 'read', AutoTimeLimit);

voltageDisplay = sprintf('Voltage : %d v', volts);
disp(voltageDisplay);
```

```
Voltage : 5 v
```

Clear the Connection

When you are finished working with the instrument, disconnect from and delete the MATLAB Instrument Object.

```
disconnect(ictObj);
delete(ictObj);
clear ictObj;
```

Using a NI Switch Module and a NI DMM to Perform Resistance Measurements

A switch module consists of an array of controllable relays/switches that can be arranged and addressed to provide various connection configurations. When used with an appropriate terminal block, an NI switch module can be configured to use a certain switch topology, such as multiplexer or matrix topology with one- or multi-wire switching modes. One common application is to combine one or more measuring instruments with a switch module to perform automated measurements on a number of devices or circuit test-points. This example uses an NI switch module together with an NI digital multimeter (DMM) to measure the resistance of 5 resistive devices in two-wire resistance measurement mode.

Hardware Setup

Note: The Switch module and DMM mentioned below are configured as simulated devices in NI-MAX for the purpose of publishing this example.

- NI 2530 switch module, used with a matrix terminal block accessory, and configured in NI MAX to use a 2-wire 4x16 matrix topology.
- NI 4065 digital multimeter (DMM) connected to switch matrix row 'r0'.
- Five resistors connected to switch matrix columns 'c0' to 'c4'.

Requirements

To run this example the following hardware support packages need to be installed:

- NI-Switch support package
- NI-DMM support package

Initialize Connection to NI Switch Module

To connect to the switch module, use the pre-built MATLAB® instrument driver 'niswitch.mdd' as installed with the NI-SWITCH support package, and as the resource name use the device name from NI MAX (such as 'PXI1Slot3'). MATLAB will use the NI-SWITCH driver to communicate with the instrument. By defaut, NI switch modules use the topology configuration specified in NI MAX (vendor-provided utility).

```
mySwitch = icdevice('niswitch.mdd', 'PXI1Slot3');
connect(mySwitch);
```

Initialize Connection to NI Digital Multimeter

To connect to the DMM instrument, use the pre-built MATLAB instrument driver 'nidmm.mdd' as installed with the NI-DMM support package. For the resource name, use the device name from NI MAX (such as 'Dev1'). MATLAB will use the NI-DMM driver to communicate with the instrument.

```
myDMM = icdevice('nidmm.mdd', 'Dev1');
connect(myDMM);
```

Define Constants used by the Driver Functions

NI-Switch and NI-DMM drivers use predefined constants as function arguments for configuring specific settings and operations. Their values are listed in the corresponding driver help files. These constants are defined in C header files included with the NI-SWITCH and NI-DMM drivers:

- 'niswitch.h' and 'IviSwtch.h' in 'C:\Program Files\IVI Foundation\IVI\Include'
- 'nidmm.h' and 'IviDmm.h' in 'C:\Program Files\IVI Foundation\IVI\Include'

```
const.NISWITCH_VAL_BREAK_BEFORE_MAKE = 1;
const.NISWITCH_VAL_SOFTWARE_TRIG = 3;
const.NISWITCH_VAL_NONE = 0;
const.NIDMM_VAL_2_WIRE_RES = 5;
const.NIDMM_VAL_SOFTWARE_TRIG = 3;
const.NIDMM_VAL_AUTO_RANGE_ON = -1;
const.NIDMM_VAL_TIME_LIMIT_AUTO = -1;
```

Reset Switch

Reset switch to default connection state.

```
invoke(mySwitch.Utility, 'Reset');
```

Configure Digital Multimeter

Configure DMM measurement type to be a two-wire resistance measurement. Configure DMM to perform an auto-range operation before each measurement, and specify the measurement resolution.

```
invoke(myDMM.ConfigurationMeasurementOptions, 'ConfigurePowerLineFrequency', 60);
MeasurementFunction = const.NIDMM_VAL_2_WIRE_RES;
Range = const.NIDMM_VAL_AUTO_RANGE_ON;
ResolutionDigits = 5.5;
invoke(myDMM.Configuration, 'ConfigureMeasurementDigits', MeasurementFunction, Range, ResolutionD
```

Make a Switch Connection and Perform a Single Resistance Measurement

Connect a path between a device under test (DUT) at switch matrix column 'c0' and measuring instrument, in this case a DMM at row 'r0'.

```
invoke(mySwitch.Route, 'Connect', 'r0', 'c0');
MaximumTimeMs = 5000;
invoke(mySwitch.Route, 'WaitForDebounce', MaximumTimeMs);
```

After switch has settled, proceed with measurement.

```
MaximumTime = const.NIDMM_VAL_TIME_LIMIT_AUTO;
reading = invoke(myDMM.Acquisition, 'Read', MaximumTime);

% Check if DMM reading was over-range
IsOverRange = invoke(myDMM.Acquisition, 'IsOverRange', reading);
disp(reading);
disp(IsOverRange);
```

```
   5.0002e+07

     0
```

Disconnect the path between terminals 'r0' and 'c0'.

```
invoke(mySwitch.Route, 'disconnect', 'r0', 'c0');
```

Software-Controlled Switch Scanning Operation

A switch module can be configured to switch between different connection paths by scanning, i.e. sequentially switching and advancing through a list of connections as defined in a scan list. The

connection switching during scanning can be synchronized with a measuring instrument by hardware-triggering (hardware-timed handshaking), or can be software-timed by a software trigger command.

To configure a scanning operation, specify a list of connections as a scan list string. Here, the switch will cycle through connections between the measuring instrument (DMM) at matrix row 'r0' and devices under test at matrix columns 'c0' to 'c4'. For details on the scan list syntax refer to the NI-SWITCH driver documentation.

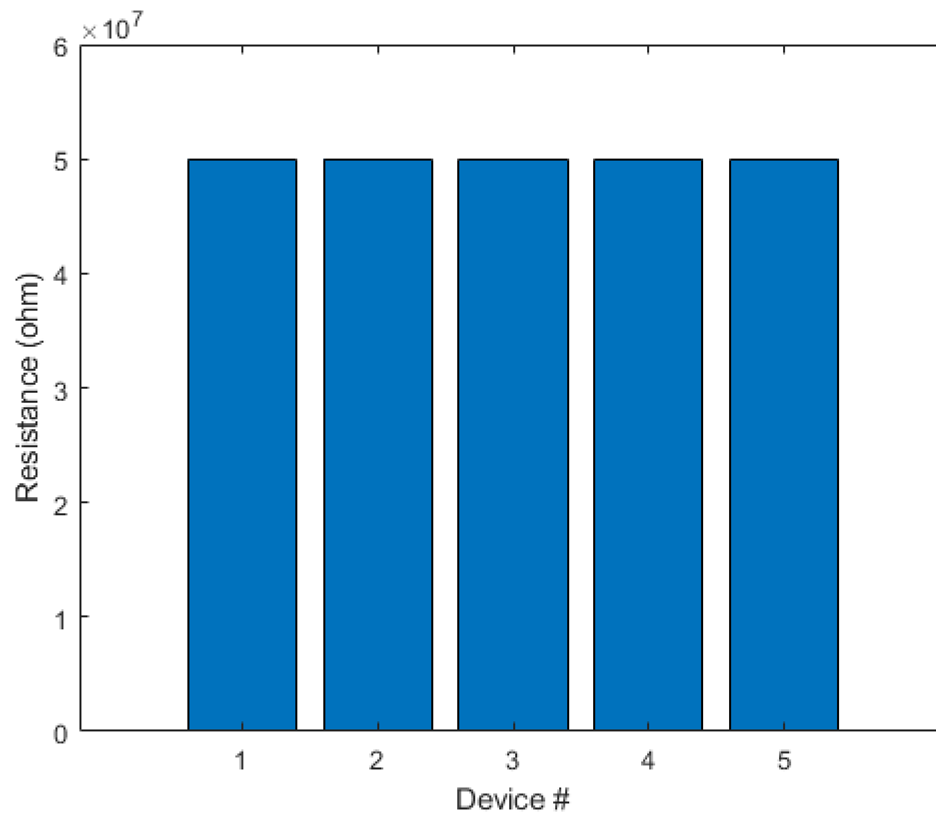
```
NrConnections = 5;
scanList = 'c0:4->r0;';
scanMode = const.NISWITCH_VAL_BREAK_BEFORE_MAKE;
invoke(mySwitch.Scan, 'configureScanList', scanList, scanMode);

% Set scan trigger input to software -- switching to the next connection in
% the scan list will be done as a result of a SendSoftwareTrigger command.
scanDelay = 0;
triggerInput = const.NISWITCH_VAL_SOFTWARE_TRIG;
scanAdvancedOutput = const.NISWITCH_VAL_NONE;
invoke(mySwitch.Scan, 'ConfigureScanTrigger', scanDelay, triggerInput, scanAdvancedOutput);

% Configure switch to cycle only once through the scan list by disabling
% the continuous scan mode.
invoke(mySwitch.Scan, 'SetContinuousScan', 0);
```

Initiate switch scan operation using software triggering.

```
invoke(mySwitch.Scan, 'InitiateScan');

for ii=1:NrConnections
    % Perform DMM measurement
    readings(ii) = invoke(myDMM.Acquisition, 'Read', const.NIDMM_VAL_TIME_LIMIT_AUTO);
    % Check if DMM reading is over-range
    IsOverRange = invoke(myDMM.Acquisition, 'IsOverRange', readings(ii));
    if IsOverRange
        fprintf('Measurement %d is over range.\n', ii);
    end
    % Send trigger command to switch to advance to next connection
    invoke(mySwitch.Scan, 'SendSoftwareTrigger');
end
```

Display measurement results.

```
figure;
bar(readings);
ylabel('Resistance (ohm)');
xlabel('Device #');
set(gca, 'XTick', [1:NrConnections]);
```



Disconnect from Switch Module and DMM Instrument

```
disconnect(mySwitch);
delete(mySwitch);

disconnect(myDMM);
delete(myDMM);
```

Generate DC Voltage Using NI-DCPOWER MATLAB Instrument Driver in Simulation Mode

This example shows how to generate DC voltage from a National Instruments® NI-DCPOWER driver in the simulation mode.

Requirements

This example requires a Microsoft® Windows® system and NI-DCPOWER package 1.7 or higher.

Verify NI-DCPOWER Installation

Use the `instrhwinfo` command to check if the NI-DCPOWER software package is installed correctly. If installed correctly, NI-DCPOWER is listed as one of the modules installed on the Windows machine. This example uses libraries installed with it.

```
driversInfo = instrhwinfo ('ivi');
disp(driversInfo.Modules');
```

```
    {'nidcpower'       }
    {'nidmm'           }
    {'niFgen'          }
    {'nisACPwr'        }
    {'niScope'         }
    {'nisCounter'      }
    {'nisDCPwr'        }
    {'nisDigitizer'    }
    {'nisDmm'          }
    {'nisDownconverter'}
    {'nisFGen'         }
    {'nisPwrMeter'     }
    {'nisRFSigGen'     }
    {'nisScope'        }
    {'nisSpecAn'       }
    {'nisSwtch'        }
    {'nisUpconverter'  }
    {'niSwitch'        }
```

Create a MATLAB Instrument Object

Use the `icdevice` function to create an instrument object from the MDD which was part of the NI-DCPOWER support package, and establish a connection to the DCPOWER using that object.

The `icdevice` function takes two or more input arguments: the MDD file name, the resource name for the DCPOWER, and optional device-specific parameters.

You can get the resource name for the DCPOWER from NI Measurement and Automation Explorer. For example: A resource name of `PXI1Slot1` in NI MAX would be `DAQ::PXI1Slot1` and `Device 1` would be `DAQ::Dev1`. You can remove the `optionstring` argument and the corresponding string parameter if you have the actual hardware.

You can establish a connection to the DCPOWER using the `connect` command.

```
ictObj = icdevice('nidcpower.mdd', 'DAQ::PXI1Slot1', 'optionstring','simulate=true');
connect(ictObj);
disp(ictObj);
```

```
   Instrument Device Object Using Driver : niDCPower

   Instrument Information
      Type:               IVIInstrument
      Manufacturer:       National Instruments Corp.
      Model:              National Instruments DC Power Supplies

   Driver Information
      DriverType:         MATLAB IVI
      DriverName:         niDCPower
      DriverVersion:      1.0

   Communication State
      Status:             open
```

Configure the DCPOWER

For the purpose of this example, the DCPOWER is configured as

```
* Channel: 0

* Source Mode: Single Point

* Output Function: DC Voltage

* Voltage Level: 6V
```

Use the MATLAB Instrument Driver Editor `midedit` to view other properties and functions that allow you to configure a device. The tool shows all the properties and functions that the NI-DCPOWER software package supports.

```
channel = '0';
src = get(ictObj, 'source');

% Configure the Source mode to Single Point
sourceMode = 1020;
invoke(src, 'configuresourcemode', sourceMode);

% Set the output function to DC Voltage
outputFunction = 1006;
invoke(src, 'configureoutputfunction', channel, outputFunction);

srcDCVoltage = get(ictObj, 'sourcedcvoltage');

% Configure the Voltage level, in volts, for the output channel generation
voltageLevel = 6;
invoke(srcDCVoltage, 'configurevoltagelevel', channel, voltageLevel);
```

Start Generation and Acquisition

```
% Initiate the device to start generation
control = get(ictObj, 'control');
invoke(control, 'initiate');
```

```
% Measure voltage
measurementType = 1;
measure = get(ictObj, 'measure');
volts = invoke(measure, 'measure', channel, measurementType);
```

Display the Read Voltage

```
voltageDisplay = sprintf('Voltage : %d v', volts);
disp(voltageDisplay);
```

```
Voltage : 6 v
```

Clear the Connection

Disconnect from and delete the MATLAB Instrument Object.

```
disconnect(ictObj);
delete(ictObj);
clear ictObj;
```

Send and Receive Multicast Data Packets Using the User Datagram Protocol

This example shows how to send and receive multicast data using `udpport`.

Create udpport Instances

Create a `udpport` instance to send the multicast data.

```
uSender = udpport()
```

```
uSender = 
  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 62055
    NumBytesAvailable: 0

  Show all properties, functions
```

Create several `udpport` instances to receive this multicast data. Ensure that all these `udpport` instances bind to the same `LocalPort` with `EnablePortSharing` set to true. In this example, the `udpport` instances `uReceiver1` and `uReceiver2` are bound to `LocalPort` 3030. `uReceiver1` is a datagram type `udpport` instance and `uReceiver2` is a byte type `udpport` instance.

```
uReceiver1 = udpport("datagram", "LocalPort", 3030, "EnablePortSharing", true)
```

```
uReceiver1 = 
  UDPPort with properties:

         IPAddressVersion: "IPV4"
                LocalHost: "0.0.0.0"
                LocalPort: 3030
    NumDatagramsAvailable: 0

  Show all properties, functions
```

```
uReceiver2 = udpport("LocalPort", 3030, "EnablePortSharing", true)
```

```
uReceiver2 = 
  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 3030
    NumBytesAvailable: 0

  Show all properties, functions
```

You can also create the sender and receivers on different MATLAB® instances to communicate between multiple MATLAB instances on the same computer.

Prepare the Multicast Receivers

Set multicast on for the `udpport` multicast receivers using the `configureMulticast` function. Subscribe to the multicast address group "226.0.0.1". When the `uSender` sends data to this multicast address group, every `udpport` instance subscribed to this address receives the multicast data.

```
configureMulticast(uReceiver1, "226.0.0.1");
configureMulticast(uReceiver2, "226.0.0.1");
```

This is reflected in the `MulticastGroup` and `EnableMulticast` properties.

```
uReceiver1.MulticastGroup
```

```
ans = 
"226.0.0.1"
```

```
uReceiver1.EnableMulticast
```

```
ans = logical
   1
```

```
uReceiver2.MulticastGroup
```

```
ans = 
"226.0.0.1"
```

```
uReceiver2.EnableMulticast
```

```
ans = logical
   1
```

Send Multicast Data

The `uSender` instance sends "hello" as a string data type to the multicast address group "226.0.0.1" and the port 3030.

```
write(uSender,"hello","string","226.0.0.1",3030);
```

Receive Multicast Data

Verify that the multicast receivers `uReceiver1` and `uReceiver2` get the data. Read this data in MATLAB.

Ensure that the receivers received the multicast packets. `uReceiver1` being a datagram type `udpport` instance receives the data as a datagram. `uReceiver2` being a byte type `udpport` instance receives the data as raw bytes.

```
uReceiver1Count = uReceiver1.NumDatagramsAvailable
```

```
uReceiver1Count = 1
```

```
uReceiver2Count = uReceiver2.NumBytesAvailable
```

```
uReceiver2Count = 5
```

Read the data from the first receiver as a string, specifying the number of datagrams to read.

```
data1 = read(uReceiver1,uReceiver1Count,"string");
```

`data1` is a `udpport.datagram.Datagram` object. View the data received

```
data1.Data
```

```
ans = 
"hello"
```

Read the data from the second receiver as a string, specifying the number of bytes of data to read.

```
data2 = read(uReceiver2,uReceiver2Count,"string")
```

```
data2 = 
"hello"
```

Unsubscribe and Clear

Unsubscribe from the multicast address group.

```
configureMulticast(uReceiver1, "off");
configureMulticast(uReceiver2, "off");
```

Clear the `udpport` instances.

```
clear uReceiver1
clear uReceiver2
clear uSender
```

Communicate Between Two MATLAB Sessions Using User Datagram Protocol

This example shows how to send data over the User Datagram Protcol (UDP) between two MATLAB® sessions on the same computer using the `udpport` function.

First MATLAB session

Create udpport Instance

Create a `udpport` instance and bind to port 2020.

```
uFirst = udpport("LocalPort",2020)
```

```
uFirst =
  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 2020
    NumBytesAvailable: 0

  Show all properties, functions
```

Prepare Callback Function

Configure the callback function to read data received using the `configureCallback` function. This callback function will trigger whenever a terminator is received. The `sendAcknowledgement` callback function sends an acknowledgement string back to the `udpport` instance in the second MATLAB session when it receives data.

Set the `Terminator` property to `"CR/LF"`, to match that of the other `udpport` instance.

```
configureTerminator(uFirst, "CR/LF");
```

The `Terminator` property should now be set to `"CR/LF"`.

```
uFirst.Terminator
```

```
ans =
"CR/LF"
```

Set up the callback function `sendAcknowledgement` to trigger when the assigned terminator is received.

```
configureCallback(uFirst,"terminator",@sendAcknowledgement);
```

The `sendAcknowledgement.m` callback function is given below.

```
function sendAcknowledgement(u, ~)
% Read the data received from the other udpport instance. readline removes
% the terminator from the data read.
data = readline(u);

% Prepare the acknowledgement string.
data = "COMMAND RECEIVED - """ + data +  """. SENDING ACKNOWLEDGEMENT.";
```

```
% Send the acknowledgement string, followed by the Terminator "CR/LF", to the
% udpport instance bound to port 3030 in the first MATLAB instance.
writeline(u, data, "127.0.0.1", 3030);
end
```

The `BytesAvailableFcn` property should be now set to the callback function, and the `BytesAvailableFcnMode` set to `"terminator"`.

```
uFirst.BytesAvailableFcn
```

```
ans = function_handle with value:
    @sendAcknowledgement
```

```
uFirst.BytesAvailableFcnMode
```

```
ans = 
"terminator"
```

Second MATLAB session

Create udpport Instance

Create a `udpport` instance and bind to port 3030.

```
uSecond = udpport("LocalPort", 3030)
```

```
uSecond = 
  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 3030
    NumBytesAvailable: 0

  Show all properties, functions
```

Prepare Callback Function

Configure the callback function `readAcknowledgement` to read data received using the `configureCallback` function. This callback function is triggered when a terminator is received. The terminator value used for this example is `"CR/LF"`.

Set the `Terminator` property to `"CR/LF"` using the `configureTerminator` function.

```
configureTerminator(uSecond,"CR/LF");
```

The `Terminator` property should now be set to `"CR/LF"`.

```
uSecond.Terminator
```

```
ans = 
"CR/LF"
```

Set up the callback function `readAcknowledgement` to trigger when the assigned terminator is received.

```
configureCallback(uSecond,"terminator",@readAcknowledgement);
```

The `readAcknowledgement.m` callback function is given below.

```
function readAcknowledgement(u, ~)
% Read the acknowledgement data. readline removes the Terminator from the
% data read.
data = readline(u);

% Display the acknowledgement string read.
disp(data);
end
```

The `BytesAvailableFcn` property should be now set to the callback function, and the `BytesAvailableFcnMode` set to `"terminator"`.

```
uSecond.BytesAvailableFcn
```

```
ans = function_handle with value:
    @readAcknowledgement
```

```
uSecond.BytesAvailableFcnMode
```

```
ans = 
"terminator"
```

Send Command to First MATLAB Session

Send a "START" command to the `udpport` instance `uFirst` on the first MATLAB session using `writeline`. `uFirst` is bound to port 2020. `writeline` automatically appends the "START" command with the `Terminator` `"CR/LF"`.

```
disp("Sending Command - START");
```

```
Sending Command - START
```

```
writeline(uSecond, "START", "127.0.0.1", 2020);
```

Send another command to the same destination address and destination port. The "STOP" command is automatically appended with the `Terminator` `"CR/LF"`.

```
disp("Sending Command - STOP");
```

```
Sending Command - STOP
```

```
writeline(uSecond, "STOP");
```

Clear udpport

Pause before clearing the object for the responses to come back from the first MATLAB session.

```
pause(0.3);
```

Configure the callbacks to be off.

```
configureCallback(uSecond, "off");
```

The `BytesAvailableFcn` property should be now set to an empty `function_handle`, and the `BytesAvailableFcnMode` set to `"off"`.

```
uSecond.BytesAvailableFcn
```

```
ans = 

  0×0 empty function_handle array
```

```
uSecond.BytesAvailableFcnMode
```

```
ans = 
"off"
```

Clear `udpport` instance.

```
clear uSecond
```

Broadcast User Datagram Protocol Data Packets

This example shows how to send and receive broadcast datagram packets using the `udpport` function.

Create udpport Broadcaster

Create a datagram type `udpport` broadcaster instance

```
uBroadcaster = udpport("datagram")
```

```
uBroadcaster = 
  UDPPort with properties:

         IPAddressVersion: "IPV4"
                LocalHost: "0.0.0.0"
                LocalPort: 59646
    NumDatagramsAvailable: 0

  Show all properties, functions
```

Set the `EnableBroadcast` property to allow for broadcasting.

```
uBroadcaster.EnableBroadcast = true;
```

Create udpport Receivers

Create `udpport` instances that receive the broadcast data. These receivers are bound to `LocalPort` 2020 with `EnablePortSharing` enabled so that multiple `udpport` objects can bind to the same socket. `uReceiver1` is a byte type `udpport` instance and `uReceiver2` is a datagram type `udpport` instance.

```
uReceiver1 = udpport("byte","LocalPort",2020,"EnablePortSharing",true)
```

```
uReceiver1 = 
  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 2020
    NumBytesAvailable: 0

  Show all properties, functions
```

```
uReceiver2 = udpport("datagram", "LocalPort",2020,"EnablePortSharing",true)
```

```
uReceiver2 = 
  UDPPort with properties:

         IPAddressVersion: "IPV4"
                LocalHost: "0.0.0.0"
                LocalPort: 2020
    NumDatagramsAvailable: 0

  Show all properties, functions
```

Send Broadcast Data

The broadcaster sends data to the broadcast address "192.168.255.255" and the port 2020, to which the receivers are bound. In this example, the broadcast address is "192.168.255.255", which is determined by the network address and the subnet mask. This address will be different on your computer.

Write the data 1:5, specified as uint8 data.

```
write(uBroadcaster,1:5,"uint8","192.168.255.255",2020);
```

Receive Broadcast Data

Now that the broadcaster has sent the data, the receivers receive these data packets.

Verify that the value of the `NumBytesAvailable` property of `uReceiver1` is 5, indicating that five bytes of data was received.

```
uReceiver1Count = uReceiver1.NumBytesAvailable
```

```
uReceiver1Count = 5
```

Verify that the value of the `NumDatagramsAvailable` property of `uReceiver2` is 1, indicating that one datagram was received.

```
uReceiver2Count = uReceiver2.NumDatagramsAvailable
```

```
uReceiver2Count = 1
```

Read the 5 bytes of data from `uReceiver1`.

```
data1 = read(uReceiver1,uReceiver1Count,"uint8")
```

```
data1 = 1×5

     1     2     3     4     5
```

Read the 1 datagram received on `uReceiver2.`

```
data2 = read(uReceiver2,uReceiver2Count,"uint8");
```

`data2` is a `udpport.datagram.Datagram` object. View the data received.

```
data2.Data
```

```
ans = 1×5

     1     2     3     4     5
```

Clear Instances

Clear the `udpport` broadcaster and receiver instances.

```
clear uBroadcaster
clear uReceiver1
clear uReceiver2
```

Communicate Binary and ASCII Data to an Echo Server Using TCP/IP

This example shows how to set up an echo server and communicate with it using TCP/IP by creating a `tcpclient` object. Binary data and terminated string data are sent to the server and the server echoes the same data back to the client.

Set Up TCP/IP Echo Server and Client

Create a TCP/IP echo server on port 4500.

```
echotcpip("on",4500);
```

Create a `tcpclient` object and connect to the server. Specify the remote host as `"localhost"` to connect to the echo server. Specify the same remote port number you used for the echo server.

```
t = tcpclient("localhost",4500)
```

```
t = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4500
    NumBytesAvailable: 0

  Show all properties, functions
```

Write and Read Binary Data Using Byte Callback Mode

Create a callback function called `readDataFcn` to read data each time the specified bytes of data are available. Store the read data in the `UserData` property of `tcpclient` object. See the `readDataFcn` function at the end of this example.

Set the callback function to trigger each time 10 bytes of data are received.

```
configureCallback(t,"byte",10,@readDataFcn);
```

Send 10 bytes of data to the echo server.

```
sendData = 1:10;
write(t,sendData,"uint8");
```

The echo server sends the binary data back to the TCP/IP client.

Pause for 1 second to allow the callback function `readDataFcn` to complete its operation.

```
pause(1);
```

Read binary data stored in `UserData` property and display it.

```
data = t.UserData
```

```
data = 1×10 uint8 row vector

    1    2    3    4    5    6    7    8    9   10
```

This data matches the data you wrote to the echo server.

Write and Read ASCII Data Using Terminator Callback Mode

Create a callback function called `readASCIIFcn` to read data each time a terminator is found in the data. Store the read data in the `UserData` property of `tcpclient` object. See the `readASCIIFcn` function at the end of this example.

Set the callback function to read terminated string data. The callback is triggered when it receives a terminator in the data.

```
configureCallback(t,"terminator",@readASCIIFcn);
```

Set the `Terminator` property value to `"LF"`.

```
configureTerminator(t,"LF");
```

Send string data to the echo server using `writeline`. The terminator character `"LF"` is automatically appended to this string data.

```
writeline(t,"Echo this string.");
```

The echo server sends the ASCII data back to the TCP/IP client.

Pause for 1 second to allow the callback function `readASCIIFcn` to complete its operation.

```
pause(1);
```

Read ASCII data stored in `UserData` property and display it.

```
textData = t.UserData
```

```
textData =
"Echo this string."
```

This data matches the data you wrote to the echo server.

Clear the Connection

Stop the echo server and clear the `tcpclient` object.

```
echotcpip("off");
clear t
```

Callback Functions

Callback Function to Read Binary Data

This function calls `read` to read `BytesAvailableFcnCount` number of bytes of data. This data is echoed back by the server.

```
function readDataFcn(src, ~)
src.UserData = read(src,src.BytesAvailableFcnCount,"uint8");
end
```

Callback Function to Read ASCII Data

This function calls `readline` to read ASCII data originally sent by the `tcpclient` object. The data is echoed back by the server.

```
function readASCIIFcn(src, ~)
src.UserData = readline(src);
end
```

Communicate Between a TCP/IP Client and Server in MATLAB

This example shows how to use the `tcpserver` and `tcpclient` functions to create a TCP/IP client and TCP/IP server in MATLAB and then send data between them over the TCP/IP protcol. You can run this example three different ways:

- Within a single MATLAB session.
- Between two MATLAB sessions on the same computer.
- Between two MATLAB sessions on different computers that are part of the same subnet.

To run this example in a single MATLAB session, you do not have to make any modifications. Create both the server and client in the same MATLAB session.

To run this example in two different MATLAB sessions either on the same computer or on two different computers, you have to run the specified sections in each MATLAB session. Copy the **Server Session** and **Callback Functions** sections to one MATLAB script in the first MATLAB session. Copy the **Client Session** section to another MATLAB script in the second MATLAB session. Run the **Server Session** script first and then run the **Client Session** script, since you must create the server before a client can attempt a connection to the server.

When using two MATLAB sessions, copy the values of `server.ServerAddress` and `server.ServerPort` from **Server Session** and use them as the `Address` and `Port` values for creating the `tcpclient` object in the **Client Session**.

Server Session

In this session, create a `tcpserver` object that listens for client connection requests. It sends data after a client connects to it. It also uses the callback functionality enabled by the `configureCallback` method to read data sent by the client.

Find Host Name and Address

Find the host name and address of the machine where the server is created. The client uses this address to connect to the server.

```
[~,hostname] = system('hostname');
hostname = string(strtrim(hostname));
address = resolvehost(hostname,"address");
```

Create Server

Create the `tcpserver` object using the address of the machine and port 5000. Create a callback function called `connectionFcn` to write data when a TCP/IP client connects to the server. Set the `ConnectionChangedFcn` property to the callback function `connectionFcn`. You can find the `connectionFcn` function at the end of this example.

```
server = tcpserver(address,5000,"ConnectionChangedFcn",@connectionFcn)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "172.28.200.248"
           ServerPort: 5000
            Connected: 0
        ClientAddress: ""
```

```
          ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Read Binary Data Using Byte Callback Mode

Create a callback function called `readDataFcn` to read data each time the specified bytes of data are available. Store the read data in the `UserData` property of `tcpserver` object. You can find the `readDataFcn` function at the end of this example.

Set the callback function to trigger each time 7688 bytes of data are received.

```
configureCallback(server,"byte",7688,@readDataFcn);
```

Client Session

In this session, create a `tcpclient` object to connect to the server. The client reads data sent from the server. It then sends the data it read back to the server.

Create Client

Create a `tcpclient` instance and set the timeout to five seconds.

When using two MATLAB sessions, copy the values of `server.ServerAddress` and `server.ServerPort` from **Server Session** and use them as the `Address` and `Port` values for creating the `tcpclient` object.

```
client = tcpclient(server.ServerAddress,server.ServerPort,"Timeout",5)
```

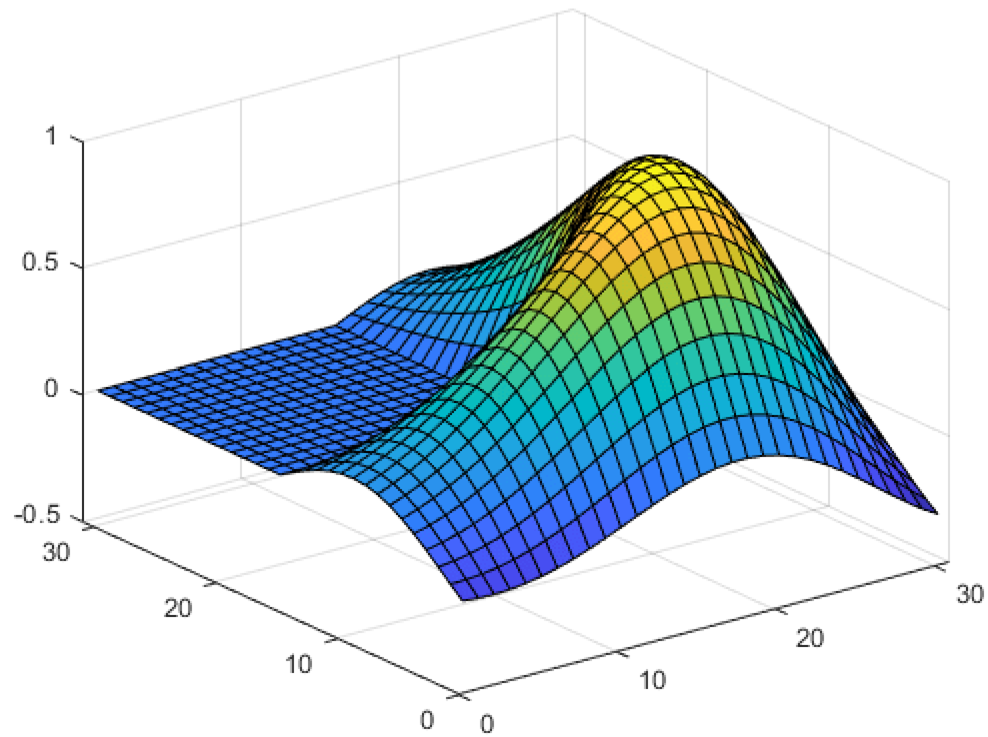
```
client = 
  tcpclient with properties:

              Address: '172.28.200.248'
                 Port: 5000
    NumBytesAvailable: 0

  Show all properties, functions
```

```
pause(1);
```

Read Data and Display

Read data sent by the server. Reshape the data array and plot it.

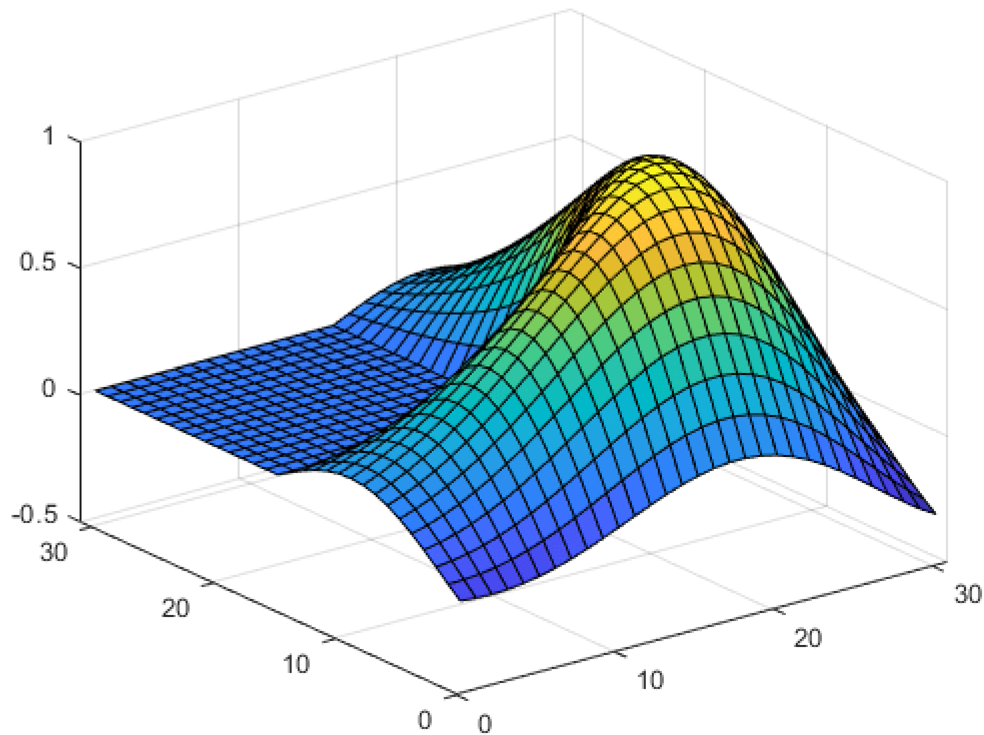
```
rawData = read(client,961,"double");
reshapedData = reshape(rawData,31,31);
surf(reshapedData);
```



Write Data

Write data to the server.

```
write(client,rawData,"double");
```



Clear the Client

Clear the `tcpclient` instance.

```
clear client
```

Callback Functions

Connection Callback Function to Write Binary Data

This function calls `write` to write data to the connected TCP/IP client.

```
function connectionFcn(src, ~)
if src.Connected
    disp("Client connection accepted by server.")
    data = membrane(1);
    write(src,data(:),"double");
end
end
```

Data Available Callback Function to Read Binary Data

This function calls `read` to read `BytesAvailableFcnCount` number of bytes of data.

```
function readDataFcn(src, ~)
disp("Data was received from the client.")
src.UserData = read(src,src.BytesAvailableFcnCount/8,"double");
reshapedServerData = reshape(src.UserData,31,31);
```

```
surf(reshapedServerData);
end
```

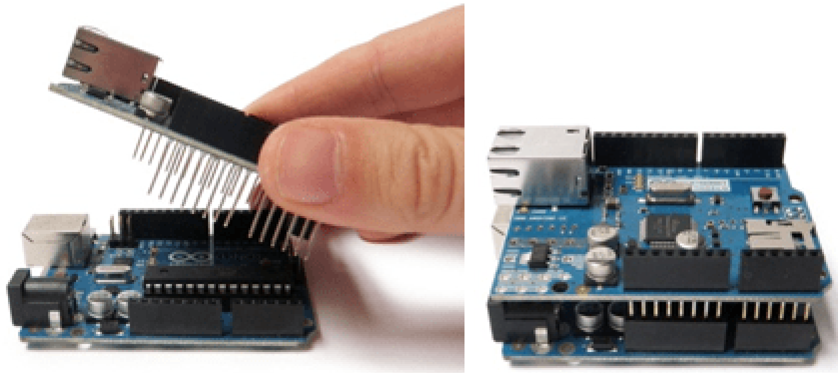
Read Data from Arduino Using TCP/IP Communication

This example shows how to enable callbacks to read sine wave data from an Arduino® Uno using the `tcpserver` interface. The Arduino is configured as a TCP/IP client and connects to the TCP/IP server created in MATLAB® using `tcpserver`.

Connect the Ethernet Shield to Arduino Uno

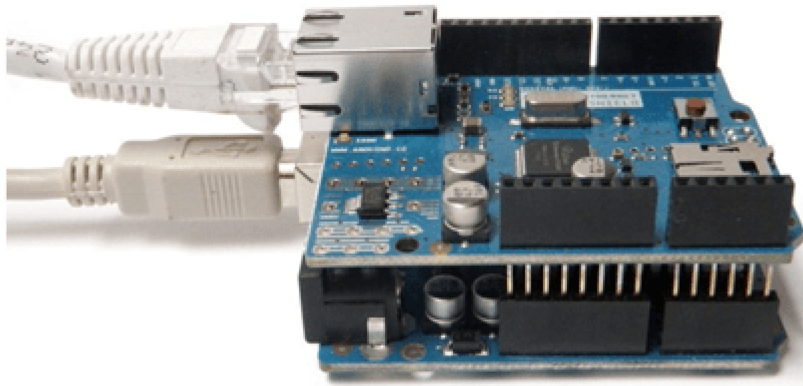
Plug in an Arduino Uno to your computer. Follow these steps to connect the W5100 Ethernet Network Shield to the Arduino Uno and to your network router or a network adapter on the computer.

Place the Ethernet Shield firmly on the Arduino Uno.



Use an RJ45 cable to connect the Arduino Ethernet Shield to one of the following:

- Network router that provides Internet to your computer.
- Network adapter on your computer.



Identify the IP address of the router or network adapter that the Arduino Ethernet Shield is connected to. Specify this IP address in the Arduino program in the **Load Program on the Arduino Uno** section. You also use this IP address as an input argument for `tcpserver` in the **Create the Server** section.

Load Program on the Arduino Uno

Load the following program on the Arduino Uno using the Arduino IDE. This program writes out 250 float values of a sine wave.

```
/*
 TCPIPClient
 Write sine wave data values to the tcpserver object created in MATLAB.
 */

#include <SPI.h>
#include <Ethernet.h>

// Specify the MAC address printed on the Ethernet shield.
// If no MAC address is printed, then use the address shown below.
byte mac[] = {0xDE,0xAD,0xBE,0xEF,0xFE,0xED};

// Specify the server IP address that is used to create the tcpserver object in MATLAB.
// This is the IP address of the router or network adapter that the Arduino Ethernet Shield is co
// In this example, 192.168.1.81 is the IP address for the server.
IPAddress server(192,168,1,81);

// Set the static IP address for the Arduino Ethernet Shield to act as a TCP/IP client.
// Choose an IP address that is in the same subnet or private network as the server IP address.
// In this example, 192.168.1.177 is the IP address for the Arduino Ethernet Shield. It is in the
IPAddress ip(192,168,1,177);
IPAddress myDns(192,168,1,1);

// Ethernet client library.
EthernetClient client;

// Command sent by the server.
byte command;

// Sine wave data buffer.
float sineWaveBuffer[250];

// The setup routine runs once when you press reset.
void setup()
{
  // Initialize serial communication.
  Serial.begin(9600);
  while (!Serial)
  {
    ; // Wait for serial port to connect.
  }

  Ethernet.begin(mac,ip,myDns);
  Serial.print("Manually assigned the following IP address to the Arduino:");
  Serial.println();
  Serial.println(Ethernet.localIP());

  // Check for Ethernet hardware.
  if (Ethernet.hardwareStatus() == EthernetNoHardware)
  {
    Serial.println("Ethernet shield was not found.");
  }
```

```
  // Check for Ethernet cable connection.
  if (Ethernet.linkStatus() == LinkOFF)
  {
    Serial.println("Ethernet cable is not connected.");
  }

  Serial.print("Attempting connection to ");
  Serial.print(server);
  Serial.println("...");

  // Attempt to connect to the server running at IP address 192.168.1.81 and port 5000.
  if (client.connect(server,5000))
  {
    Serial.print("Connected to server running at ");
    Serial.println(client.remoteIP());
  }
  else
  {
    Serial.println("Connection to server failed.");
  }

  // Store sine wave data as 250 float values.
  for (int j = 0;j < 250;j++)
  {
    sineWaveBuffer[j] = sin(j*50.0/360.0);
  }
}

// Main processing loop
void loop()
{
  // Block until data is sent by server.
  if (client.available() > 0)
  {
    // Read the command sent by the server.
    command = client.read();

    // Print the command sent by the server.
    Serial.println("The server sent the following command:");
    Serial.println(command);

    if (client.connected() && command == 1)
    {
        // Write sine wave data to the server.
        client.write((const uint8_t *) & sineWaveBuffer, sizeof(sineWaveBuffer));
    }
  }
}
```

Create the Server

Create a `tcpserver` instance using the IP address of the router or network adapter.

In this example, the IP address is `192.168.1.81` and the port number is `5000`. This IP address must be the same one specified in the Arduino program.

```
server = tcpserver("192.168.1.81",5000)
```

```
server =
  TCPServer with properties:

        ServerAddress: "192.168.1.81"
           ServerPort: 5000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Prepare the `tcpserver` Object to Receive Data

Set the `ConnectionChangedFcn` property to `@requestDataCommand.` The callback function `requestDataCommand` is triggered when the Arduino connects to the `tcpserver` object.

```
server.ConnectionChangedFcn = @requestDataCommand;
```

Create a callback function `requestDataCommand` that sends uint8 value of 1 as a command to request the Arduino to send data.

```
function requestDataCommand(src,~)
if src.Connected
    % Display the server object to see that Arduino client has connected to it.
    disp("The Connected and ClientAddress properties of the tcpserver object show that the Ardui
    disp(src)

    % Request the Arduino to send data.
    disp("Send the command: 1")
    write(src,1,"uint8");
end
end
```

Set the `BytesAvailableFcnMode` property to "`byte`", the `BytesAvailableFcn` property to `@readArduinoData,` and the `BytesAvailableFcnCount` property to 1000.

```
configureCallback(server,"byte",1000,@readArduinoData);
```

The callback function `readArduinoData` is triggered when 250 sine wave float data points (1000 bytes) are available to be read from the Arduino.

Read Callback Function

Create a callback function `readArduinoData` that reads 250 sine wave data points and plots the result.

```
function readArduinoData(src,~)
% Read the sine wave data sent to the tcpserver object.
src.UserData = read(src,src.BytesAvailableFcnCount/4,'single');

% Plot the data.
plot(src.UserData)
end
```

Generate a Swept Sinusoid Using VISA and Capture Waveform Using Quick-Control Oscilloscope

This example shows how to use a function generator to generate a swept sinusoid waveform and also how to capture it using an oscilloscope.

For a complete list of supported hardware, visit the Instrument Control Toolbox product page.

Requirements

This example was tested using a Keysight Technologies® 33522B function generator and a Tektronix® TDS 1002 oscilloscope. The GPIB addresses of the function generator and oscilloscope are **GPIB0::5::INSTR** and **GPIB0::11::INSTR**, respectively. The function generator is configured to generate a 2V p-p swept-sinusoid (20 to 200 Hz) with an offset of 1V every 100 ms on channel 1. The oscilloscope is configured to acquire a waveform on channel 1.

Configure the Oscilloscope

Configure the oscilloscope using a Quick-Control (oscilloscope).

```
scopeResource = "GPIB0::11::INSTR";
ch = "CH1";
```

Create an oscilloscope object and open a connection to the instrument.

```
scope = oscilloscope;
scope.Resource = scopeResource;
connect(scope)
```

The `autoSetup` function automatically adjust channels, vertical, horizontal, and trigger controls based on connected signals.

```
autoSetup(scope)
```

Enable and configure channel 1.

```
enableChannel(scope,ch);
configureChannel(scope,ch,"VerticalCoupling","DC")
```

Configure the channel to display at 1 `VOLTS/DIV.`

```
configureChannel(scope,ch,"VerticalRange",1)
```

Set probe attenuation to 1x (options include 1, 10, 100).

```
configureChannel(scope,ch,"ProbeAttenuation",1)
```

`The AcquisitionTime` property represents the waveform duration in seconds. Setting the `AcquisitionTime` will change the `SEC/DIV` control accordingly. `AcquisitionTime` typically corresponds to 10 divisions (or one screen of data).

```
scope.AcquisitionTime = 0.25;

scope.TriggerLevel = 2.56;
scope.TriggerSource = ch;
scope.TriggerSlope = "rising";
```

```
scope.TriggerMode = "normal";

disp(scope)
```

```
oscilloscope: TEKTRONIX,TDS 1002

   Instrument Settings:
      AcquisitionStartDelay: 'Not supported'
            AcquisitionTime: 0.25 s
               ChannelNames: 'CH1', 'CH2', 'MATH', 'REFA', 'REFB'
            ChannelsEnabled: 'CH1'
            SingleSweepMode: 'off'
                    Timeout: 10 s
             WaveformLength: 2500

   Trigger Settings:
               TriggerLevel: 2.56
              TriggerSource: 'CH1'
               TriggerSlope: 'rising'
                TriggerMode: 'normal'

   Communication Properties:
                     Status: 'open'
                   Resource: 'GPIB0::11::INSTR'

lists of methods
```

Configure the Function Generator

Configure the function generator to generate a sweep waveform using a VISA-GPIB object.

```
fgenResource = "GPIB0::5::INSTR";
vfgen = visadev(fgenResource)
```

```
vfgen = 
  GPIB with properties:

         ResourceName: "GPIB0::5::INSTR"
                Alias: "FGEN_2CH"
               Vendor: "Agilent Technologies"
                Model: "33522B"
           BoardIndex: 0
       PrimaryAddress: 5
     SecondaryAddress: 65535
    NumBytesAvailable: 0

  Show all properties, functions
```

Configure the sweep amplitude and offset.

```
writeline(vfgen,"SOUR1:VOLT +1.0")
writeline(vfgen,"SOUR1:VOLT:OFFS +1.0")
```

Enable sweep mode.

```
writeline(vfgen,"SOUR1:FREQ:MODE SWE");
writeline(vfgen,"SOUR1:SWE:STAT ON");
writeline(vfgen,"SOUR1:SWE:SPAC LIN");
```

Configure the start and stop frequencies.

```
fstart = 20;
fstop = 200;

writeline(vfgen,compose("SOUR1:FREQ:STAR %d",fstart));
writeline(vfgen,compose("SOUR1:FREQ:STOP %d",fstop));
```

Configure the time taken to sweep from the start frequency to the stop frequency as `sweepTime`.

```
sweepTime = 0.1;
holdTime = 0;
returnTime = 0;

writeline(vfgen,compose("SOUR1:SWE:TIME %0.1f",sweepTime));
```

Configure time to remain at the stop frequency as `holdTime`.

```
writeline(vfgen,compose("SOUR1:SWE:HTIME %0.1f",holdTime));
```

Configure the time required to return to the start frequency as `returnTime`.

```
writeline(vfgen,compose("SOUR1:SWE:RTIME %0.1f",returnTime));
```

Configure the trigger.

```
writeline(vfgen,"TRIG1:SLOP POS");
writeline(vfgen,"TRIG1:SOUR IMM");
```

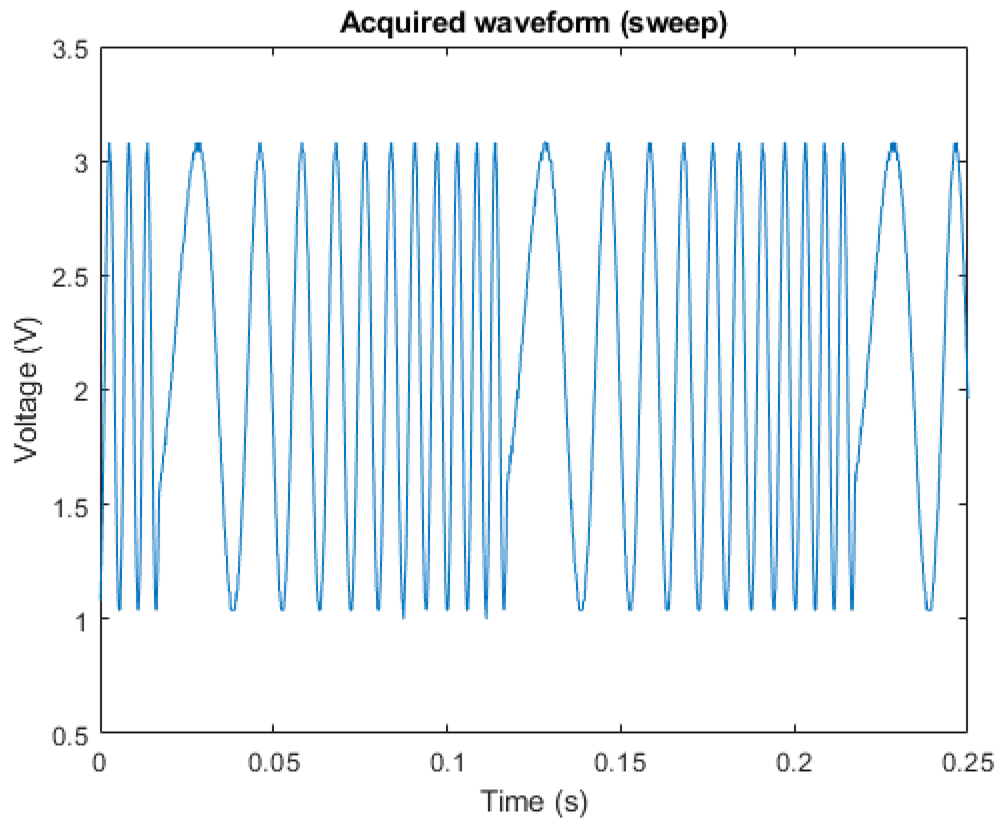
Acquire the Waveform

Acquire waveform data using the oscilloscope. The `AcquisitionTime` property represents the waveform duration in seconds. The `WaveformLength` property represents the number of points in the waveform data.

```
y = readWaveform(scope);
t = linspace(0,scope.AcquisitionTime,scope.WaveformLength);
```

Plot the Waveform

```
plot(t,y)
ylim([0.5,3.5]);
title("Acquired waveform (sweep)")
xlabel("Time (s)");
ylabel("Voltage (V)");
```



Clean Up

Clear the workspace when you are finished.

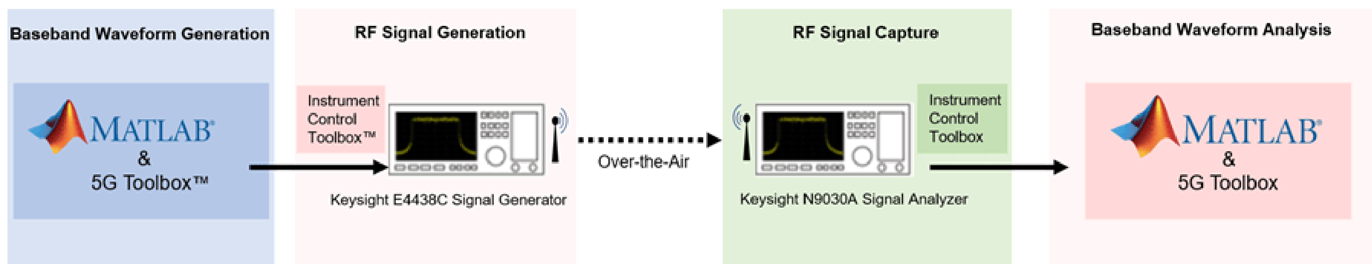
```
disconnect(scope)
clear scope vfgen
```

5G NR Waveform Acquisition and Analysis

This example shows how to generate a 5G NR test model (NR-TM) waveform using the 5G Waveform Generator (5G Toolbox) app and download the generated waveform to a Keysight™ vector signal generator for over-the-air transmission using the Instrument Control Toolbox™ software. The example then captures the transmitted over-the-air signal using a Keysight signal analyzer and analyzes the signal in MATLAB®.

Introduction

This example generates a 5G NR-TM waveform using the **5G Waveform Generator** app, downloads and transmits the waveform onto a Keysight vector signal generator, and then receives the waveform using a Keysight signal analyzer for waveform analysis in MATLAB. This diagram shows the general workflow.



Requirements

To run this example, you need these instruments:

- Keysight E4438C ESG vector signal generator
- Keysight N9030A PXA signal analyzer

Generate Baseband Waveform Using 5G Waveform Generator App

In MATLAB, on the **Apps** tab, click the **5G Waveform Generator** app.

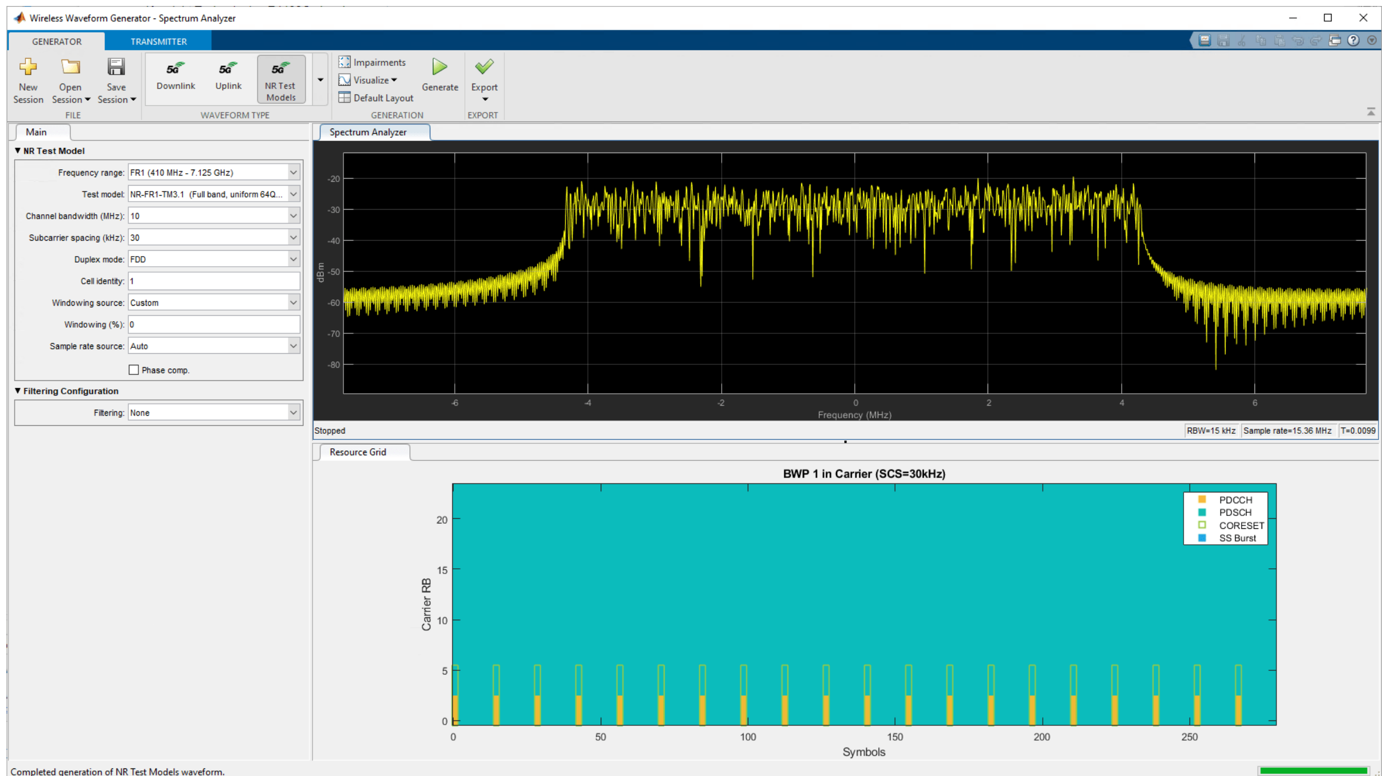
In the **Waveform Type** section, click **NR Test Models.** In the left-most pane of the app, you can set the parameters for the selected waveform. For this example:

- Set **Frequency range** as `FR1 (410 MHz - 7.125 GHz)`
- Set **Test model** as `NR-FR1-TM3.1 (Full band, uniform 64 QAM)`
- Set **Channel bandwidth (MHz)** as `10`
- Set **Subcarrier spacing (kHz)** as `30`
- Set **Duplex mode** as `FDD`.

On the app toolstrip, click **Generate**.

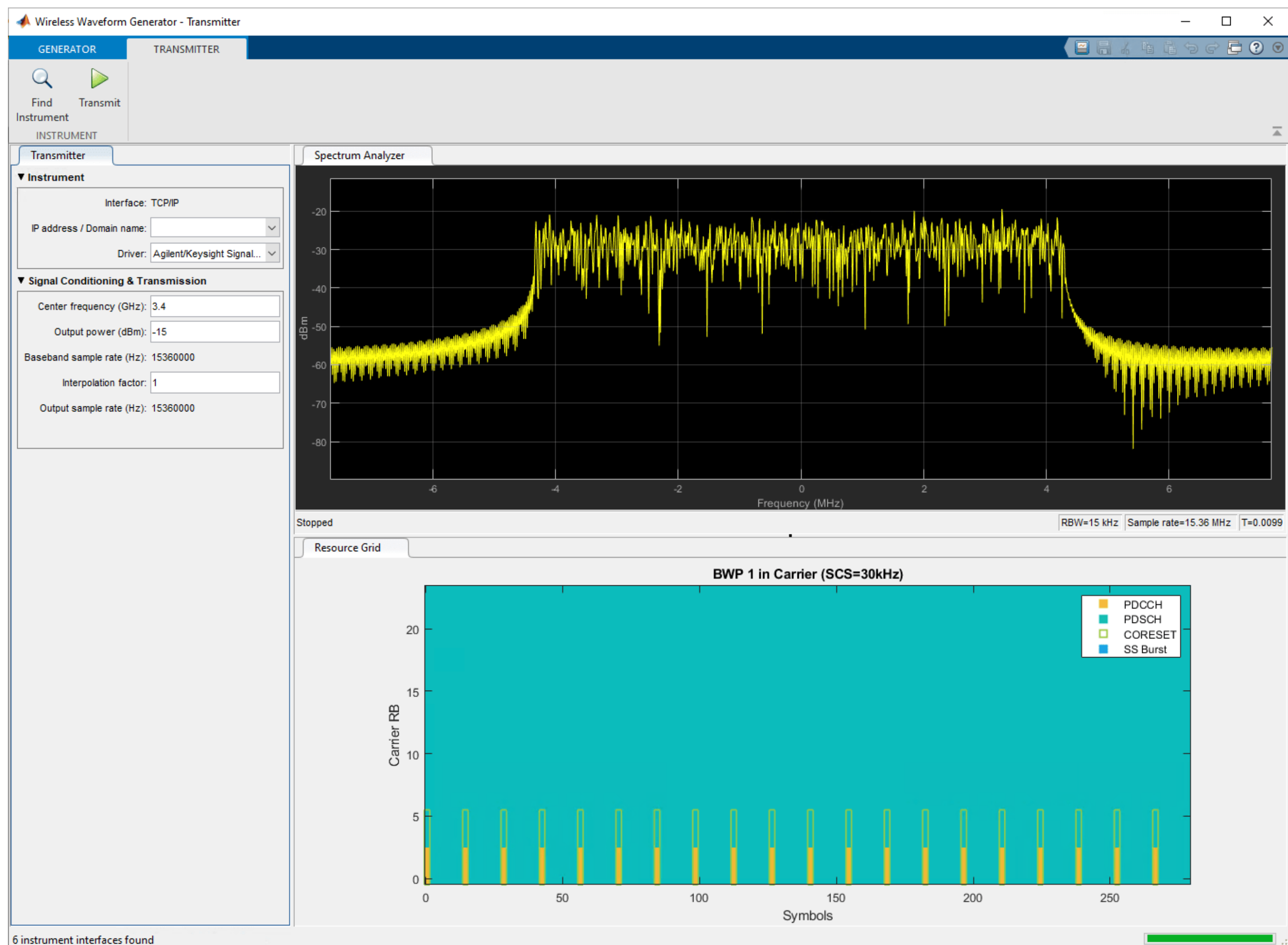
```
% Set the NR-TM parameters for the receiver
nrtm = "NR-FR1-TM3.1"; % Reference channel
bw   = "10MHz";        % Channel bandwidth
scs  = "30kHz";        % Subcarrier spacing
dm   = "FDD";          % Duplexing mode
```

This figure shows a 10 MHz 5G NR waveform visible at baseband.



Transmit Over-the-Air Signal

Download the generated signal to the RF signal generator over one of the supported communication interfaces (requires Instrument Control Toolbox). The app automatically finds the signal generator that is connected over the TCP/IP interface. On the **Transmitter** tab of the app, select `Agilent/Keysight Signal Generator SCPI` from the **Driver** list. Set the **Center frequency (GHz)** parameter to `3.4` and the **Output power (dBm)** parameter to `-15`. The app automatically obtains the baseband sample rate from the generated waveform. To start the transmission, click **Transmit** in the toolstrip.



Read IQ Data from a Signal Analyzer over TCP/IP

To read the in-phase and quadrature (IQ) data into MATLAB for analysis, configure the Keysight N9030A signal analyzer using the Instrument Control Toolbox software.

Define the instrument configuration parameters based on the signal you are measuring.

```
% Set parameters for the spectrum analyzer
centerFrequency = 3.4e9;
sampleRate = 15.36e6;
measurementTime = 20e-3;
mechanicalAttenuation = 0; %dB
startFrequency = 3.39e9;
stopFrequency = 3.41e9;
resolutionBandwidth = 220e3;
videoBandwidth = 220000;
```

Perform these steps before connecting to the spectrum analyzer.

- Find the resource ID of the Keysight N9030A signal analyzer.
- Connect to the instrument using the virtual instrument software architecture (VISA) interface.

- Adjust the input buffer size to hold the data that the instrument returns.
- Set the timeout to allow sufficient time for the measurement and data transfer.

```
foundVISA = visadevlist;
resourceID = foundVISA(foundVISA.Model == "N9030A",:).ResourceName;
resourceID = resourceID(contains(resourceID,"N9030A"));            % Extract resourceID which co
sigAnalyzerObj = visadev(resourceID);
sigAnalyzerObj.ByteOrder = "big-endian";
configureCallback(sigAnalyzerObj,"byte",85e6,@callbackFcn)
sigAnalyzerObj.Timeout = 20;
```

Reset the instrument to a known state using the appropriate standard command for programmable instruments (SCPI). Query the instrument identity to ensure the correct instrument is connected.

```
writeline(sigAnalyzerObj,"*RST");
instrumentInfo = writeread(sigAnalyzerObj,"*IDN?");
fprintf("Instrument identification information: %s",instrumentInfo);
```

```
Instrument identification information: Agilent Technologies,N9030A,US00071181,A.14.16
```

The X-Series signal and spectrum analyzers perform IQ measurements as well as spectrum measurements. In this example, you acquire time domain IQ data, visualize the data using MATLAB, and perform signal analysis on the acquired data. The SCPI commands configure the instrument and define the format of the data transfer after the measurement is complete.

```
% Set up signal analyzer mode to basic IQ mode
writeline(sigAnalyzerObj,":INSTrument:SELect BASIC");

% Set the center frequency
writeline(sigAnalyzerObj,strcat(":SENSe:FREQuency:CENTer ",num2str(centerFrequency)));

% Set the capture sample rate
writeline(sigAnalyzerObj,strcat(":SENSe:WAVeform:SRATe ",num2str(sampleRate)));

% Turn off averaging
writeline(sigAnalyzerObj,":SENSe:WAVeform:AVER OFF");

% Set the spectrum analyzer to take one single measurement after the trigger line goes high
writeline(sigAnalyzerObj,":INIT:CONT OFF");

% Set the trigger to external source 1 with positive slope triggering
writeline(sigAnalyzerObj,":TRIGger:WAVeform:SOURce IMMediate");
writeline(sigAnalyzerObj,":TRIGger:LINE:SLOPe POSitive");

% Set the time for which measurement needs to be made
writeline(sigAnalyzerObj,strcat(":WAVeform:SWE:TIME ",num2str(measurementTime)));

% Turn off electrical attenuation
writeline(sigAnalyzerObj,":SENSe:POWer:RF:EATTenuation:STATe OFF");

% Set the mechanical attenuation level
writeline(sigAnalyzerObj,strcat(":SENSe:POWer:RF:ATTenuation ",num2str(mechanicalAttenuation)));

% Turn IQ signal ranging to auto
writeline(sigAnalyzerObj,":SENSe:VOLTage:IQ:RANGe:AUTO ON");

% Set the endianness of returned data
writeline(sigAnalyzerObj,":FORMat:BORDer NORMal");
```

```
% Set the format of the returned data
writeline(sigAnalyzerObj,":FORMat:DATA REAL,64");
```

Trigger the instrument to make the measurement. Wait for the measurement operation to complete, and then read-in the waveform. Before processing the data, separate the I and Q components from the interleaved data that is received from the instrument and create a complex vector in MATLAB.

```
% Trigger the instrument and initiate measurement
writeline(sigAnalyzerObj,"*TRG");
writeline(sigAnalyzerObj,":INITiate:WAVeform");

% Wait until measure operation is complete
measureComplete = writeread(sigAnalyzerObj,"*OPC?");

% Read the IQ data
writeline(sigAnalyzerObj,":READ:WAV0?");
data = readbinblock(sigAnalyzerObj,"double");

% Separate the data and build the complex IQ vector
inphase = data(1:2:end);
quadrature = data(2:2:end);
rxWaveform = inphase+1i*quadrature;
```

Capture and display the information about the most recently acquired data.

```
writeline(sigAnalyzerObj,":FETCH:WAV1?");
signalSpec = readbinblock(sigAnalyzerObj,"double");

% Display the measurement information
captureSampleRate = 1/signalSpec(1);
fprintf("Sample Rate (Hz) = %s",num2str(captureSampleRate));
```

```
Sample Rate (Hz) = 15360000
```

```
fprintf("Number of points read = %s",num2str(signalSpec(4)));
```

```
Number of points read = 307201
```

```
fprintf("Max value of signal (dBm) = %s",num2str(signalSpec(6)));
```

```
Max value of signal (dBm) = -39.0019
```

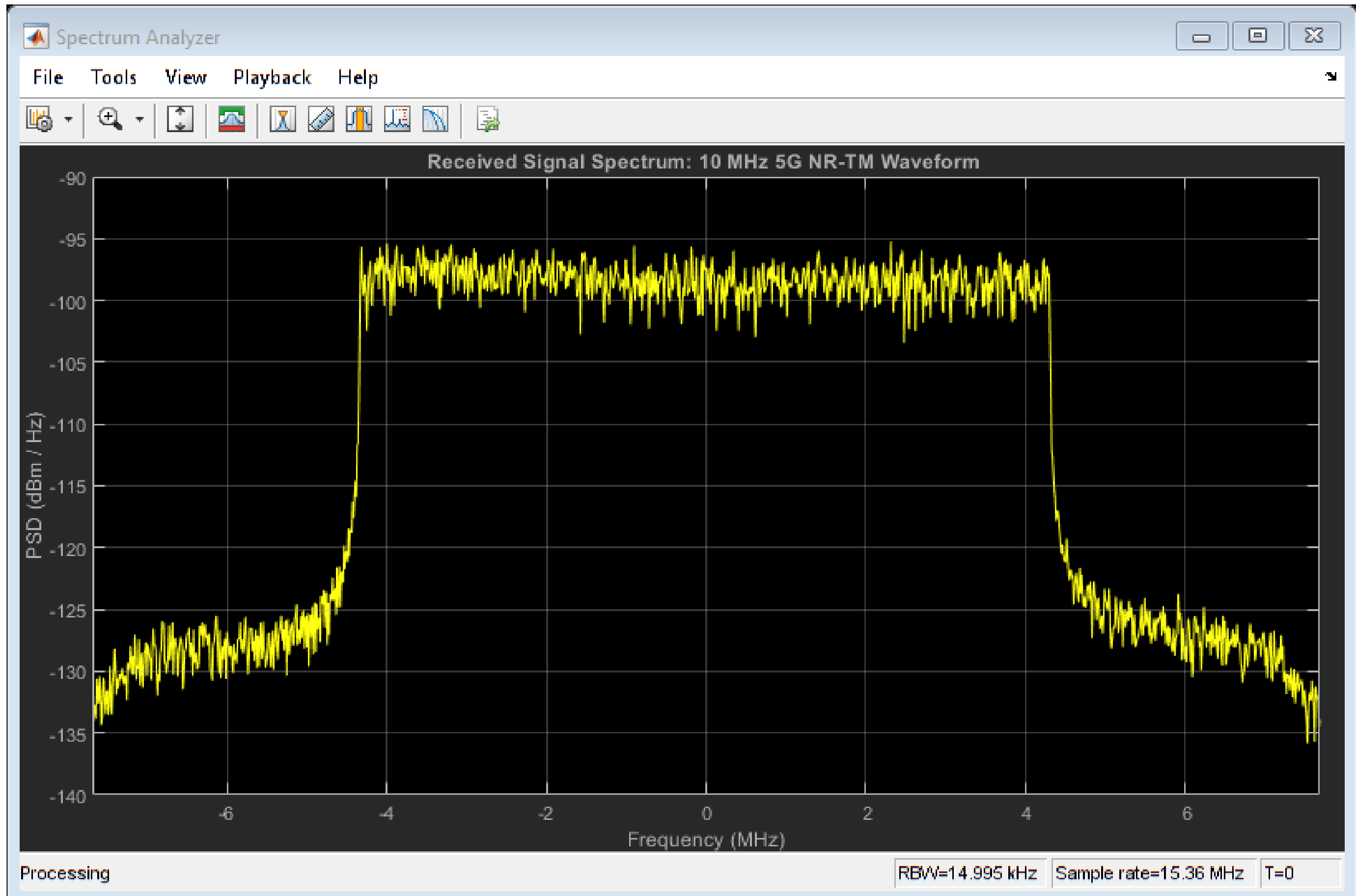
```
fprintf("Min value of signal (dBm) = %s",num2str(signalSpec(7)));
```

```
Min value of signal (dBm) = -107.9755
```

Plot the spectrum of the acquired waveform to confirm the bandwidth of the received signal.

```
% Ensure rxWaveform is a column vector
if ~iscolumn(rxWaveform)
    rxWaveform = rxWaveform.';
end

% Plot the power spectral density (PSD) of the acquired signal
spectrumPlotRx = dsp.SpectrumAnalyzer;
spectrumPlotRx.SampleRate =  captureSampleRate;
spectrumPlotRx.SpectrumType = "Power density";
spectrumPlotRx.PowerUnits =  "dBm";
```

```
spectrumPlotRx.Window = "Hamming";
spectrumPlotRx.SpectralAverages = 10;
spectrumPlotRx.YLimits = [-140 -90];
spectrumPlotRx.YLabel = "PSD";
spectrumPlotRx.ShowLegend = false;
spectrumPlotRx.Title = "Received Signal Spectrum: 10 MHz 5G NR-TM Waveform";
spectrumPlotRx(rxWaveform);
```



Switch the instrument to spectrum analyzer mode and compare the spectrum view generated in MATLAB with the view on the signal analyzer. Use additional SCPI commands to configure the instrument measurement and display settings.

```
% Switch back to the spectrum analyzer view
writeline(sigAnalyzerObj,":INSTrument:SELect SA");

% Set the mechanical attenuation level
writeline(sigAnalyzerObj,strcat(":SENSe:POWer:RF:ATTenuation ",num2str(mechanicalAttenuation)));

% Set the center frequency, RBW, and VBW
writeline(sigAnalyzerObj,strcat(":SENSe:FREQuency:CENTer ",num2str(centerFrequency)));
writeline(sigAnalyzerObj,strcat(":SENSe:FREQuency:STARt ",num2str(startFrequency)));
writeline(sigAnalyzerObj,strcat(":SENSe:FREQuency:STOP ",num2str(stopFrequency)));
writeline(sigAnalyzerObj,strcat(":SENSe:BANDwidth:RESolution ",num2str(resolutionBandwidth)));
writeline(sigAnalyzerObj,strcat(":SENSe:BANDwidth:VIDeo ",num2str(videoBandwidth)));
```

```
% Enable continuous measurement on the spectrum analyzer
writeline(sigAnalyzerObj,":INIT:CONT ON");

% Begin receiving the over-the-air signal
writeline(sigAnalyzerObj,"*TRG");
```

For instrument cleanup, clear the instrument connection:

```
clear sigAnalyzerObj;
```

To stop the 5G NR-TM waveform transmission, in the **Instrument** section on the app toolstrip, click **Stop Transmission**.

Perform Measurements of Received 5G Waveform

Use the `generateWaveform` function of the `hNRReferenceWaveformGenerator` helper file to extract the waveform information for a specific TM.

```
tmwavegen = hNRReferenceWaveformGenerator(nrtm,bw,scs,dm);
[~,tmwaveinfo,resourcesInfo] = generateWaveform(tmwavegen);
```

Coarse Frequency Offset Compensation Using Demodulation Reference Symbols (DM-RS)

Look for offsets in increments of 1 kHz up to 100 kHz.

```
frequencyCorrectionRange = -100e3:1e3:100e3;
[rxWaveform, coarseOffset] = DMRSFrequencyCorrection(rxWaveform,captureSampleRate,frequencyCorrec
fprintf("Coarse frequency offset = %.0f Hz", coarseOffset)
```

```
Coarse frequency offset = 0 Hz
```

Fine Frequency Offset Compensation Using DM-RS

Look for offsets in increments of 5 Hz up to 100 Hz

```
frequencyCorrectionRange = -100:5:100;
[rxWaveform, fineOffset] = DMRSFrequencyCorrection(rxWaveform,captureSampleRate,frequencyCorrecti
fprintf("Fine frequency offset = %.1f Hz", fineOffset)
```

```
Fine frequency offset = -30.0 Hz
```

EVM Measurements

Use the `hNRPDSCHEVM` function to analyze the waveform. The function performs these steps.

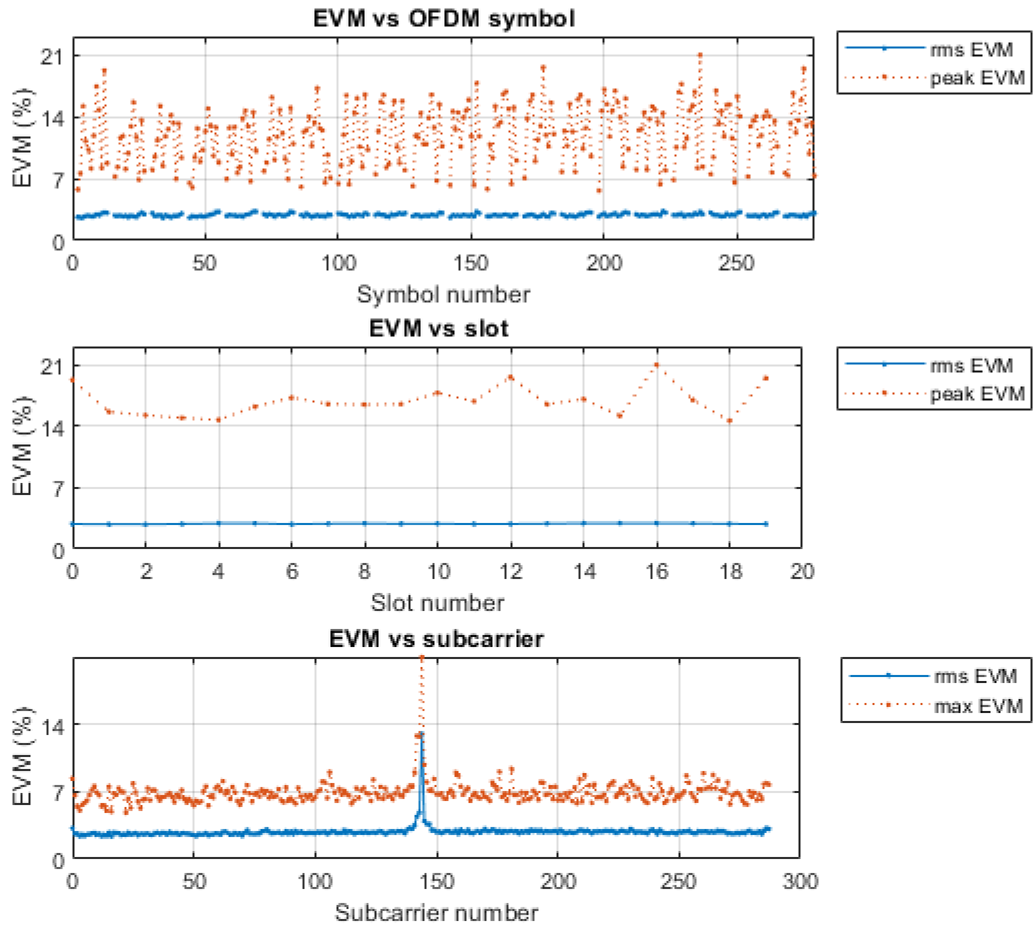
- Synchronizes the DM-RS over one frame for frequency division duplexing (FDD) (two frames for time division duplexing (TDD))
- Demodulates the received waveform
- Estimates the channel
- Equalizes the symbols
- Estimates and compensates for common phase error (CPE)

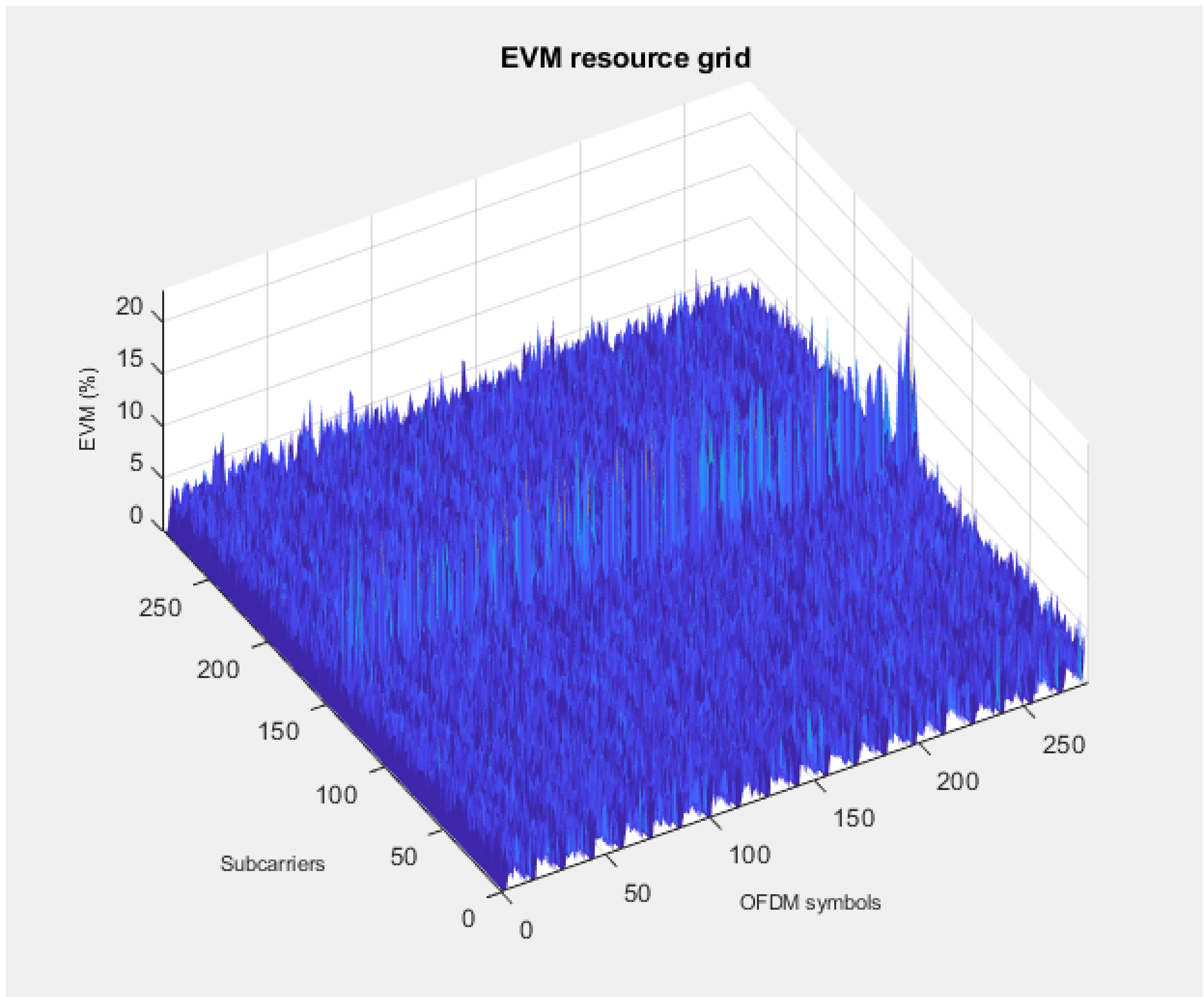
Define the configuration settings for the `hNRPDSCHEVM` function.

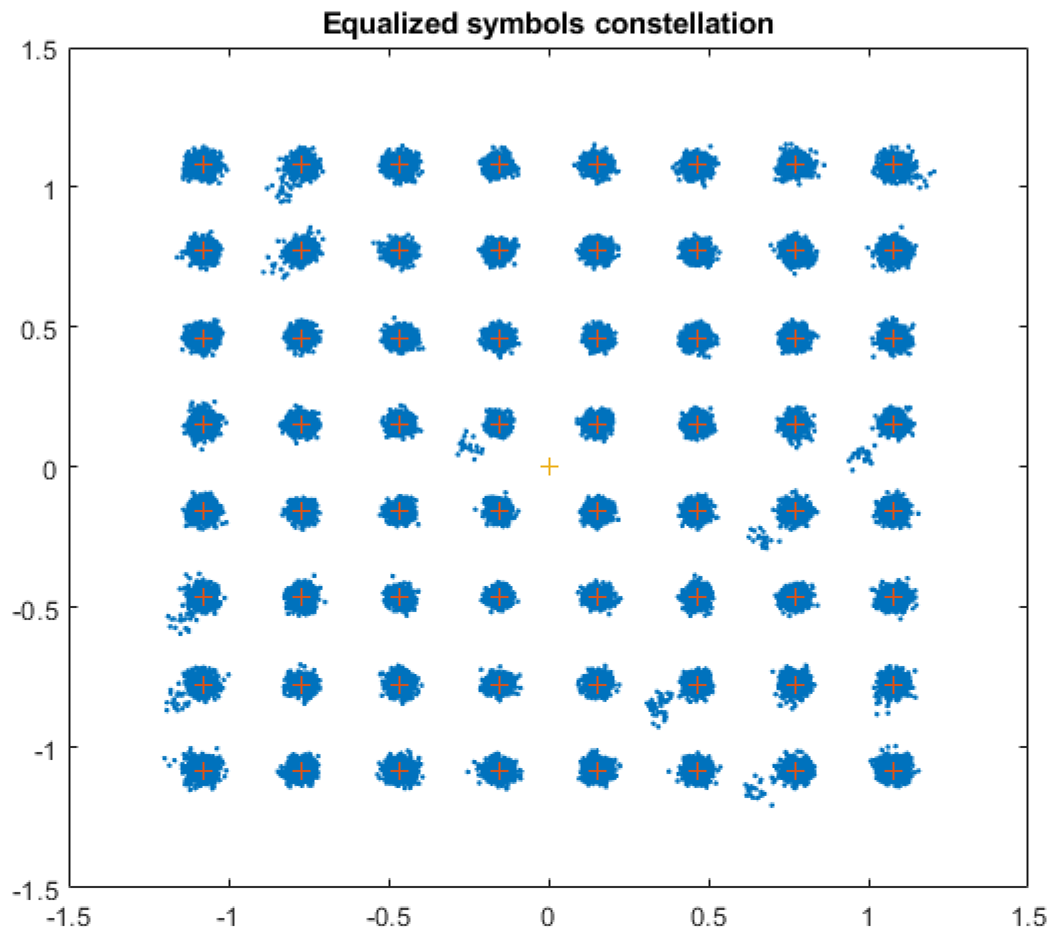
```
cfg = struct();
cfg.PlotEVM = true;                 % Plot EVM statistics
cfg.DisplayEVM = true;              % Print EVM statistics
```

```
cfg.Label = nrtm;                      % Set to TM name of captured waveform
cfg.SampleRate = captureSampleRate; % Use sample rate during capture

[evmInfo,eqSym,refSym] = hNRPDSCHEVM(tmwavegen.Config,rxWaveform,cfg);
```

```
 RMS EVM, Peak EVM, slot 0: 2.848 19.257%
 RMS EVM, Peak EVM, slot 1: 2.829 15.633%
 RMS EVM, Peak EVM, slot 2: 2.812 15.226%
 RMS EVM, Peak EVM, slot 3: 2.864 14.929%
 RMS EVM, Peak EVM, slot 4: 2.932 14.695%
 RMS EVM, Peak EVM, slot 5: 2.934 16.210%
 RMS EVM, Peak EVM, slot 6: 2.835 17.239%
 RMS EVM, Peak EVM, slot 7: 2.897 16.520%
 RMS EVM, Peak EVM, slot 8: 2.924 16.450%
 RMS EVM, Peak EVM, slot 9: 2.883 16.497%
 RMS EVM, Peak EVM, slot 10: 2.889 17.794%
 RMS EVM, Peak EVM, slot 11: 2.853 16.835%
 RMS EVM, Peak EVM, slot 12: 2.860 19.586%
 RMS EVM, Peak EVM, slot 13: 2.908 16.476%
 RMS EVM, Peak EVM, slot 14: 2.947 17.077%
 RMS EVM, Peak EVM, slot 15: 2.950 15.162%
 RMS EVM, Peak EVM, slot 16: 2.940 20.989%
 RMS EVM, Peak EVM, slot 17: 2.927 16.974%
 RMS EVM, Peak EVM, slot 18: 2.876 14.606%
 RMS EVM, Peak EVM, slot 19: 2.860 19.478%
Averaged RMS EVM frame 0: 2.889%
Averaged overall RMS EVM: 2.889%
Overall Peak EVM = 20.9891%
```


EVM vs OFDM symbol
EVM (%)
Symbol number
rms EVM
peak EVM
EVM vs slot
Slot number
EVM vs subcarrier
Subcarrier number
max EVM


EVM resource grid
EVM (%)
20
15
10
5
0
Subcarriers
250
200
150
100
50
0
OFDM symbols
0
50
100
150
200
250



The measurements show that the demodulation of the received waveform is successful. The interference from the DC component of the spectrum analyzer to the DC subcarrier causes high EVM values in the measurements.

Local functions

These functions assist in processing the received 5G waveform.

```
function [correctedWaveform,appliedFrequencyCorrection] = DMRSFrequencyCorrection(waveform,sample
    % waveform - Waveform to be corrected. Needs to be a Nx1 column vector.
    % sampleRate - Sample rate of waveform
    % frequencyCorrectioRange - Range and granularity at which frequency
    % correction is inspected
    % tmwavegen and resourcesInfo - Outputs of generateWaveform method
    [pdschArray,~,carrier] = hListTargetPDSCHs(tmwavegen.Config,resourcesInfo.WaveformResources)
    bwpCfg = tmwavegen.Config.BandwidthParts{1,1};
    nSlots = carrier.SlotsPerFrame;

    % Generate a reference grid spanning 10 ms (one frame). This grid
    % contains only the DM-RS and is used for synchronization.
    refGrid = referenceGrid(carrier,bwpCfg,pdschArray,nSlots);
```

```
    % Apply frequency offsets to the waveform as specified by
    % freuqnecyCorrectionRange.
    nSamples = (0:length(waveform)-1)';
    frequencyShift = (2*pi*frequencyCorrectionRange.*nSamples)./sampleRate;

    % Each column represents an offset waveform.
    offsetWaveforms = waveform.*exp(1j*frequencyShift);

    [~,mag] = nrTimingEstimate(offsetWaveforms,carrier.NSizeGrid,...
        carrier.SubcarrierSpacing,nSlots,refGrid, ...
        "SampleRate",sampleRate);

    % Find the frequency at which the DM-RS correlation is at a maximum.
    [~,index] = max(max(mag));
    appliedFrequencyCorrection = frequencyCorrectionRange(index);
    correctedWaveform = offsetWaveforms(:,index);
end

function refGrid = referenceGrid(carrier,bwpCfg,pdschArray,nSlots)
    % Create a reference grid for the required number of slots. The grid
    % contains the DM-RS symbols specified in pdschArray. The function
    % returns REFGRID of dimensions K-by-S-by-L, where K is the number of
    % subcarriers of size carrier.NSizeGrid, S is the number of symbols
    % spanning nSlots, and L is the number of layers.

    nSubcarriers = carrier.NSizeGrid * 12;
    L = carrier.SymbolsPerSlot*nSlots;                           % Number of OFDM symbols in the
    nLayers = size(pdschArray(1).Resources(1).ChannelIndices,2);
    bwpStart = bwpCfg.NStartBWP;
    bwpLen = bwpCfg.NSizeBWP;
    refGrid = zeros(nSubcarriers,L,nLayers);                     % Empty grid
    bwpGrid = zeros(bwpLen*12,L,nLayers);
    rbsPerSlot = bwpLen*12*carrier.SymbolsPerSlot;

    % Populate the DM-RS symbols in the reference grid for all slots. Place
    % bwpGrid in a carrier grid (at an appropriate location) in case the
    % BWP size is not the same as the carrier grid
    for slotIdx = carrier.NSlot + (0:nSlots-1)
        [~,~,dmrsIndices,dmrsSymbols] = hSlotResources(pdschArray,slotIdx);
        if ~isempty(dmrsIndices)
            for layerIdx = 1:nLayers
                if layerIdx <= size(dmrsIndices,2)
                    dmrsIndices(:,layerIdx) = dmrsIndices(:,layerIdx) - rbsPerSlot*(layerIdx -1)
                    bwpGrid(dmrsIndices(:,layerIdx)+(slotIdx-carrier.NSlot)*rbsPerSlot) = dmrsSym
                end
            end
            refGrid(12*bwpStart+1:12*(bwpStart+bwpLen),:,:) = bwpGrid;
        end
    end
end
```

TCP/IP Client Block Communication with Arduino Server

This example shows how the TCP/IP blocks in the Instrument Control Toolbox can be used with a remote server. In this example the server is running on an Arduino connected on the network. Instrument Control Toolbox provides TCP/IP blocks which are client-only blocks. Both the Send and Receive blocks can communicate via or to a server running remotely or in a different MATLAB session or to an echoserver on the same machine.

In this example there are two models which show the following:

1. Continuous stream of data from the server after a trigger.

2. On-demand read from the server after a command message is sent to the server.

Setup

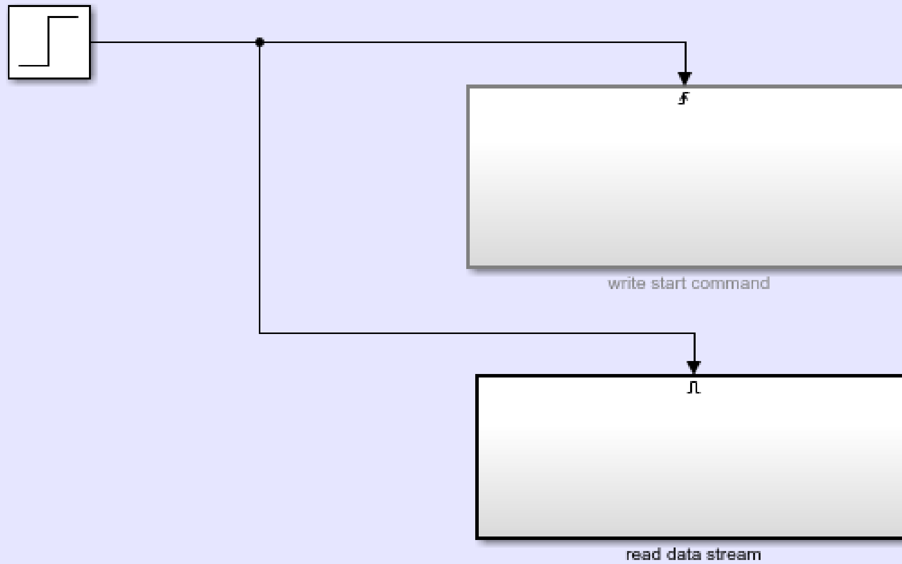
You will have to load the INO file that the example provides onto an Arduino Uno and configure your network connections to the IP and Port settings in the INO file. Running the example opens both models associated with the example. You can run one at a time to get a better understanding of the difference between the two and notice how the TCP/IP blocks behave as client blocks while communicating with the server running on the Arduino.

Continuous stream of data from the server

This model consists of two subsystems:

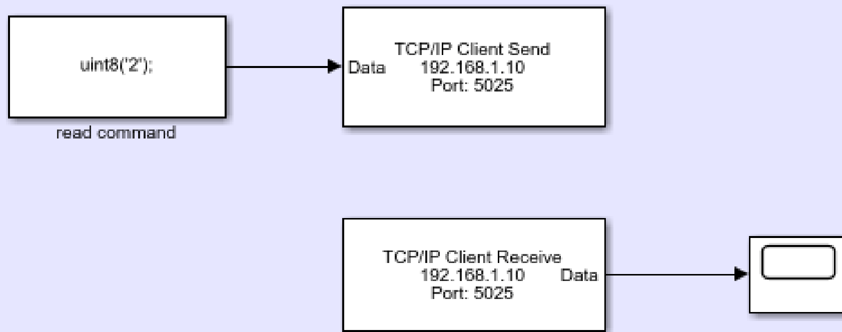
1. Write start command - This is a triggered subsystem which at the first time interval trigger sends a start command to the server running on the Arduino using the TCP/IP Send Client block.

2. Read data stream - This is a triggered subsystem which continously reads the stream of data coming in from the Arduino server at every time interval using the TCP/IP Receive Client block.



On-demand read from the server

This model is set up to send an initial send command using the TCP/IP Send Client block for the server to acknowledge that a signal has been received and for it to send some data back which is read by the TCP/IP Receive Client block.



Results

This example makes use of the sensor attached to the analog pins on the Arduino. For the type of sensor attached you should see a corresponding output.

Using the Instrument Control Toolbox Block Library

The Instrument Control Toolbox software includes a Simulink® software interface called the Instrument Control Toolbox block library. You can use the blocks of this library in a Simulink model to communicate with an instrument.

- "Open Instrument Control Toolbox Block Library" on page 23-2
- "Send and Receive Data Through Serial Port Loopback" on page 23-5
- "Send and Receive Data over TCP/IP Network" on page 23-14
- "Enable Blocking Mode in Receive and Send Blocks" on page 23-25
- "Timing in Hardware Interface Models" on page 23-30

Open Instrument Control Toolbox Block Library

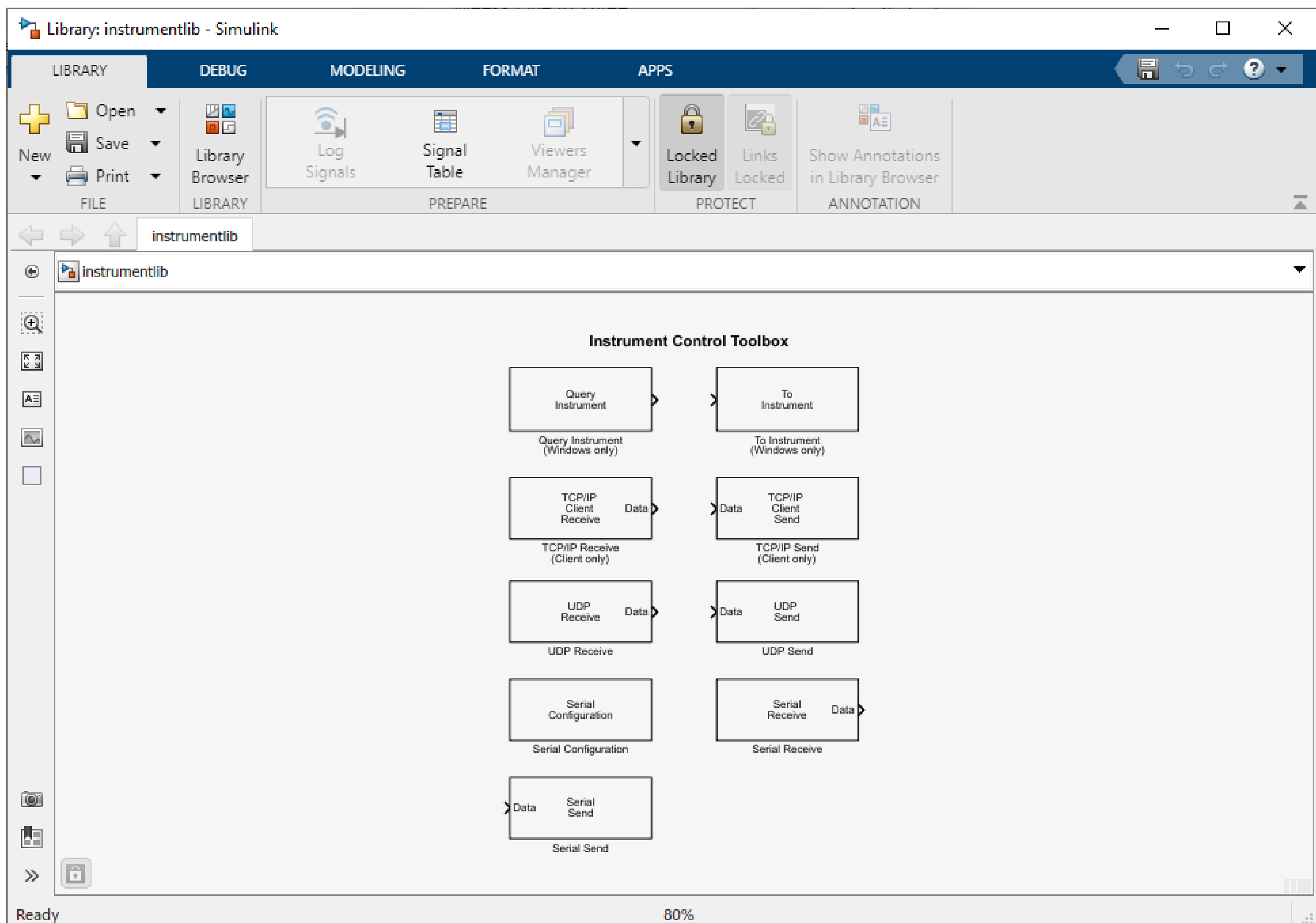
In this section...
“From the Command Line” on page 23-2
“From the Simulink Library Browser” on page 23-2

From the Command Line

To open the Instrument Control Toolbox block library, enter the following in the MATLAB Command Window.

```
instrumentlib
```

MATLAB displays the contents of the library in a separate window.

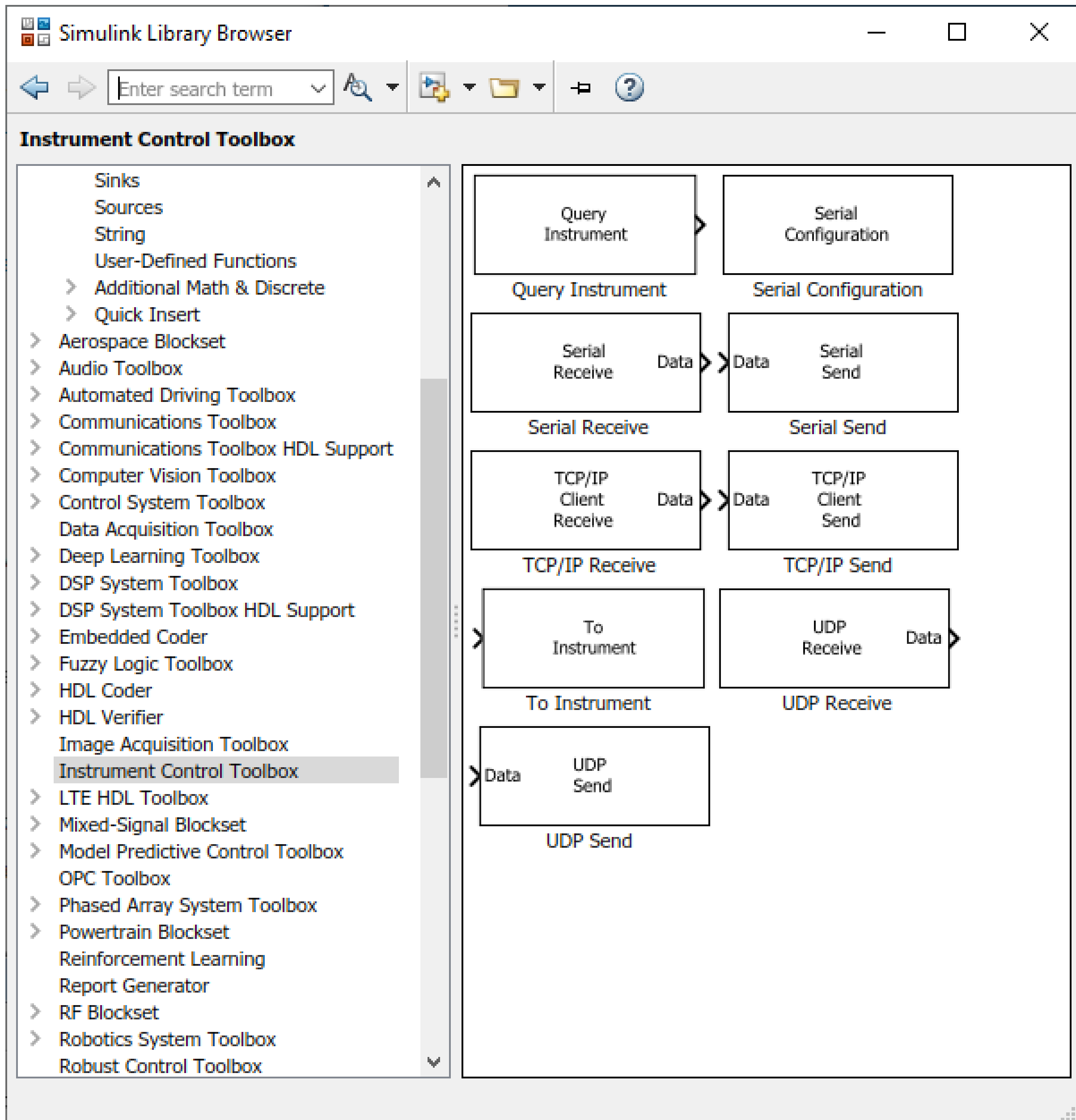


From the Simulink Library Browser

To open the Instrument Control Toolbox block library, first start the Simulink Library Browser by entering the following in the MATLAB Command Window.

```
slLibraryBrowser
```

MATLAB opens the browser window. The left pane lists available block libraries, with the basic Simulink library listed first, followed by other libraries in alphabetical order. Click **Instrument Control Toolbox**.



Simulink loads the library and displays the blocks in the library.

See Also

More About

- “Send and Receive Data Through Serial Port Loopback” on page 23-5
- “Send and Receive Data over TCP/IP Network” on page 23-14

Send and Receive Data Through Serial Port Loopback

This example shows how to build a simple model using the Instrument Control Toolbox blocks in conjunction with other blocks in the Simulink library. The example illustrates how to send data to a simple loopback device connected to the computer's serial port and how to read that data back into your model.

Note After you connect your loopback device to your computer, determine which serial port it is on. For instructions, see "Find Serial Port Information for Your Platform" on page 6-9.

You use the To Instrument block to write a value to the serial port on your computer, and then use the Query Instrument block to read that same value back into your model.

Note Block names are not shown by default in the model. To display the hidden block names while working in the model, select **Display** and clear the **Hide Automatic Names** check box.

In this section...

"Step 1: Create a New Model" on page 23-5
"Step 2: Open the Block Library" on page 23-5
"Step 3: Drag the Instrument Control Toolbox Blocks into the Model" on page 23-6
"Step 4: Drag Other Blocks to Complete the Model" on page 23-7
"Step 5: Connect the Blocks" on page 23-8
"Step 6: Specify the Block Parameter Values" on page 23-9
"Step 7: Specify the Block Priority" on page 23-12
"Step 8: Run the Simulation" on page 23-12

Step 1: Create a New Model

To start Simulink and create a new model, enter the following in the MATLAB Command Window.

```
simulink
```

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

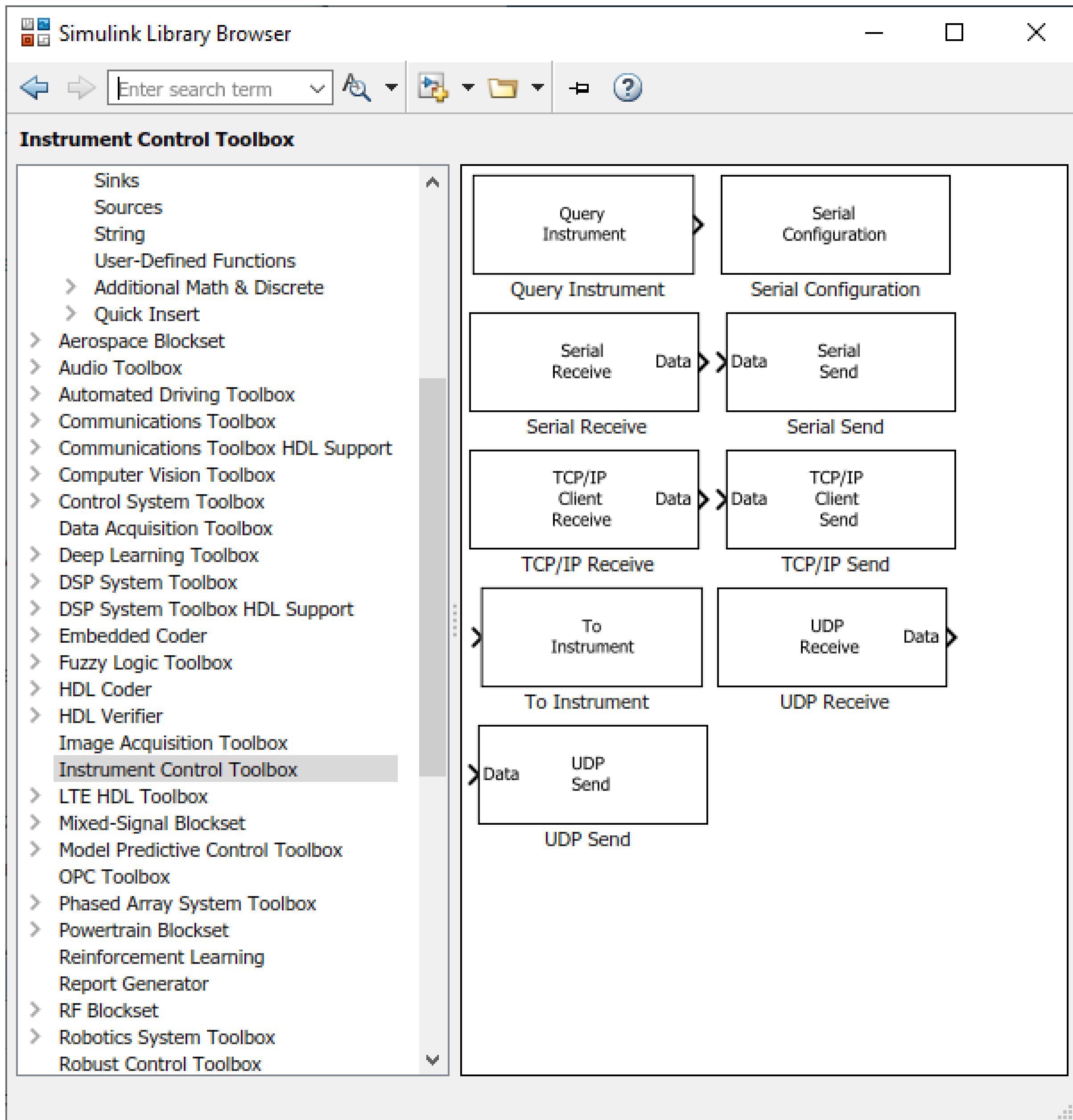
In the toolstrip, click **Save** to assign a name to your new model.

Step 2: Open the Block Library

In the toolstrip, click the **Library Browser** button in the **Simulation** tab.

The Simulink Library Browser opens. The left pane contains a tree of available block libraries in alphabetical order. Click **Instrument Control Toolbox**.

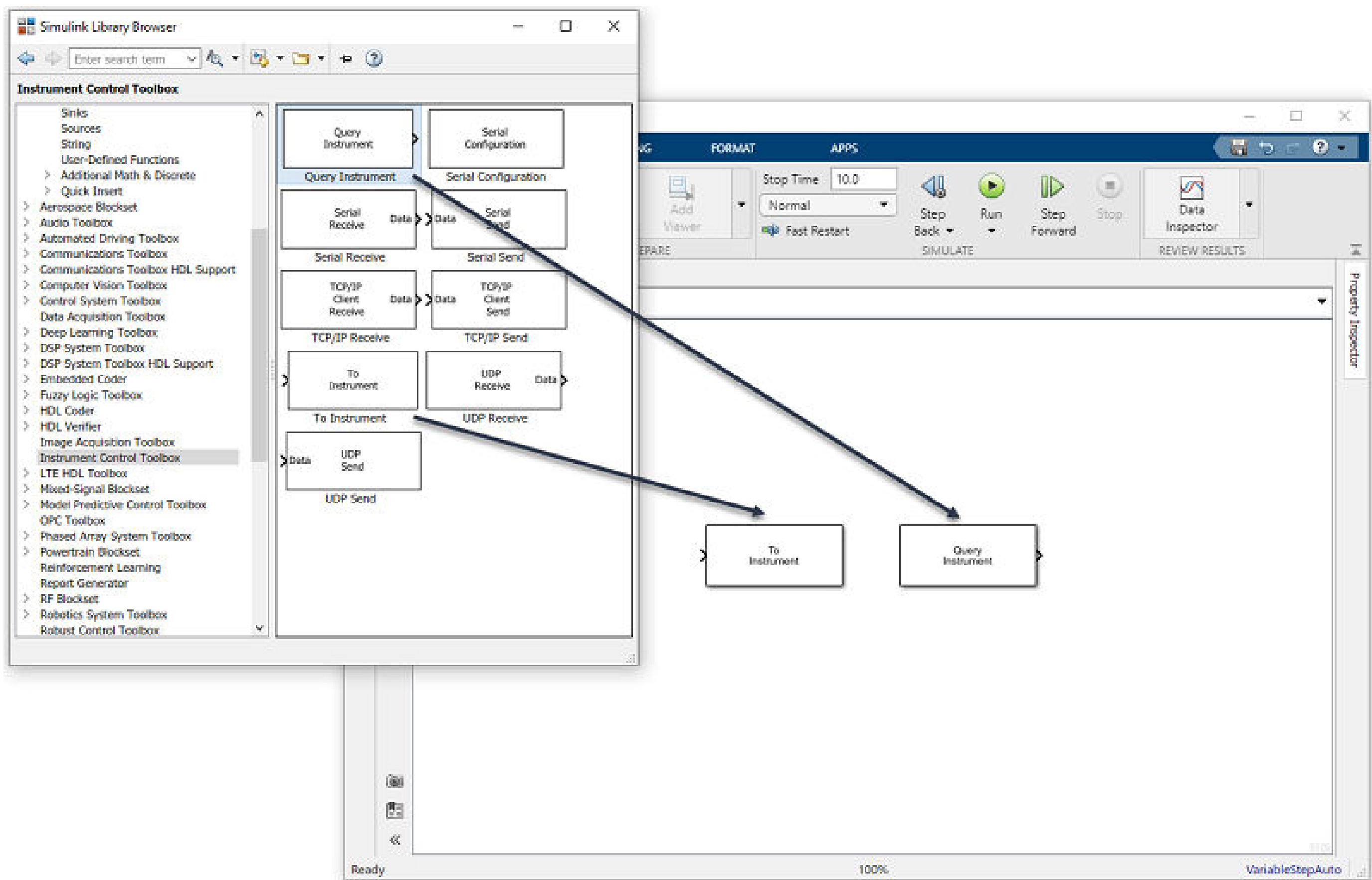
To use a block, add it to an existing model or create a new model.



Step 3: Drag the Instrument Control Toolbox Blocks into the Model

To use a block in a model, drag the block from the library into the Simulink Editor. For this example, you need one instance of the To Instrument and the Query Instrument blocks in your model.

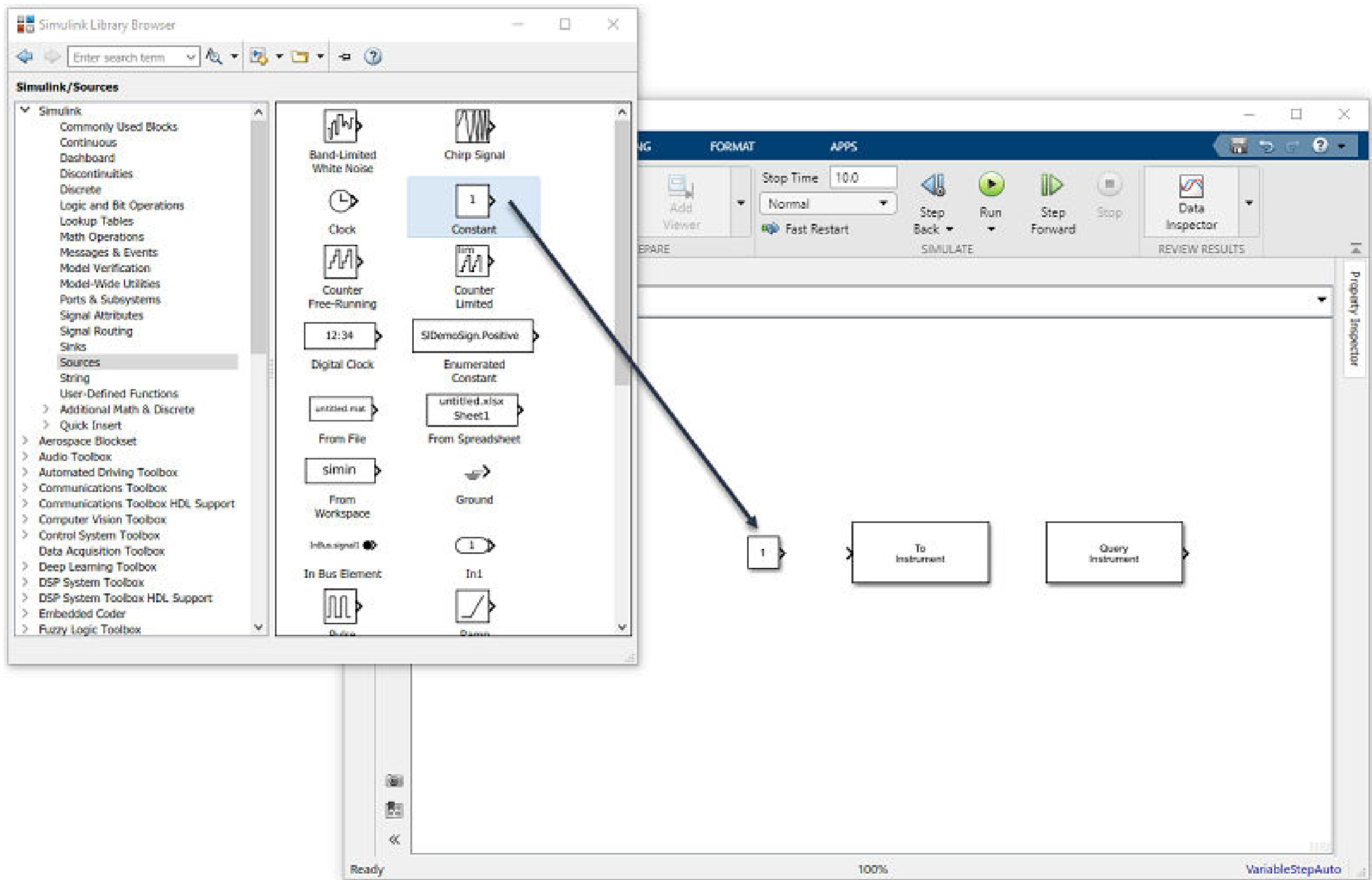
Note The To Instrument block can be used with these interfaces: VISA, GPIB, Serial, TCP/IP, and UDP. It is not supported on these interfaces: SPI, I2C, and Bluetooth.



Step 4: Drag Other Blocks to Complete the Model

This example requires two more blocks. One block provides the data that is sent to the instrument; the other block displays the data received from the instrument.

Because the data sent to the instrument is constant, you can use the Constant block for this purpose. Access the block by expanding the **Simulink** node in the browser tree and clicking the **Sources** library entry. From the blocks in the right pane, drag the Constant block into the Simulink Editor and place it to the left of the To Instrument block.

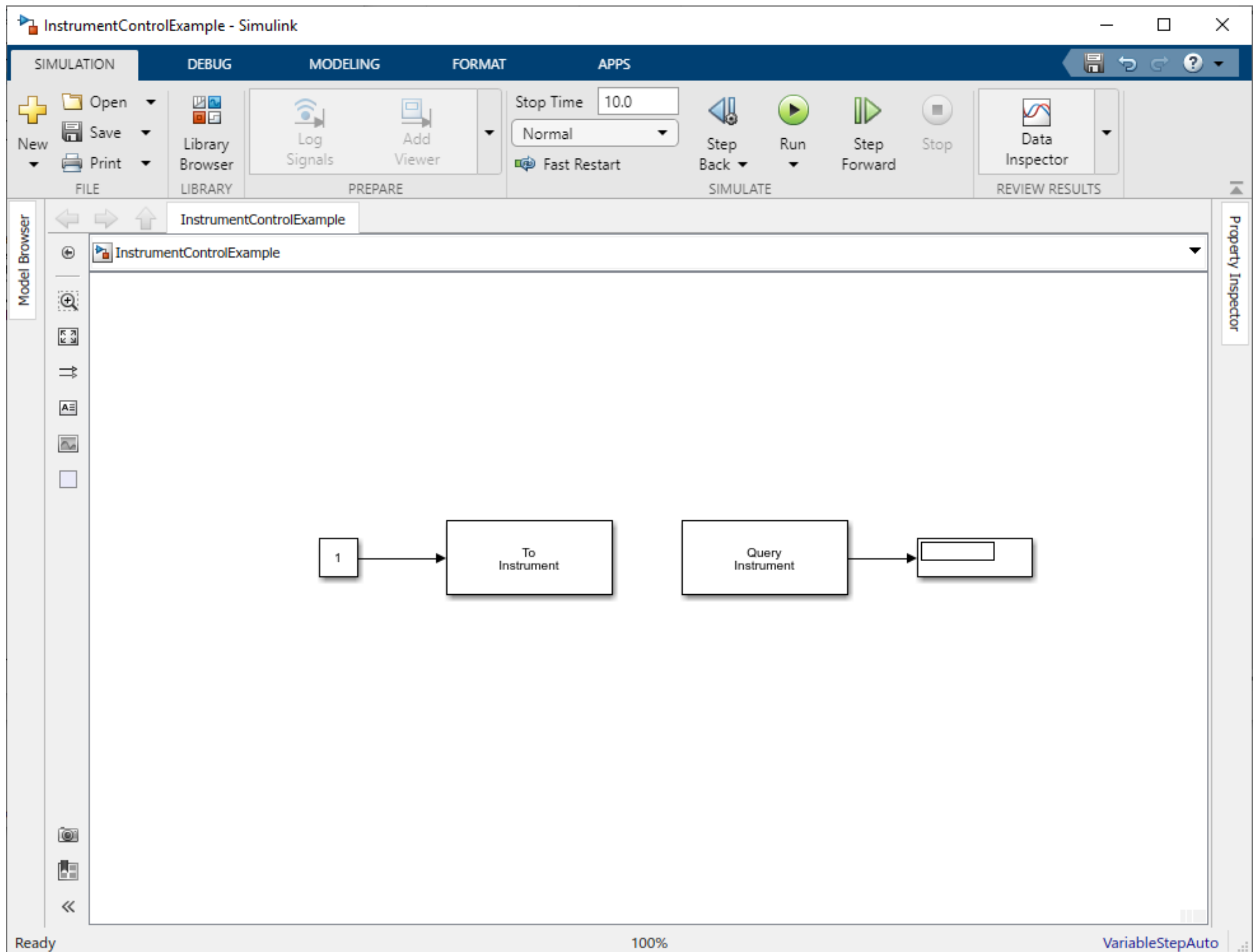


To display the data received from the instrument, you can use the Display block. To access the Display block, click the **Sinks** library entry in the expanded **Simulink** node in the browser tree. From the blocks displayed in the right pane, drag the Display block into the Simulink Editor and place it to the right of the Query Instrument block.

Step 5: Connect the Blocks

Make a connection between the Constant block and the To Instrument block. A quick way to make the connection is to select the Constant block, press and hold the **Ctrl** key, and then click the To Instrument block.

In the same way, make the connection between the output port of the Query Instrument block and the input port of the Display block.

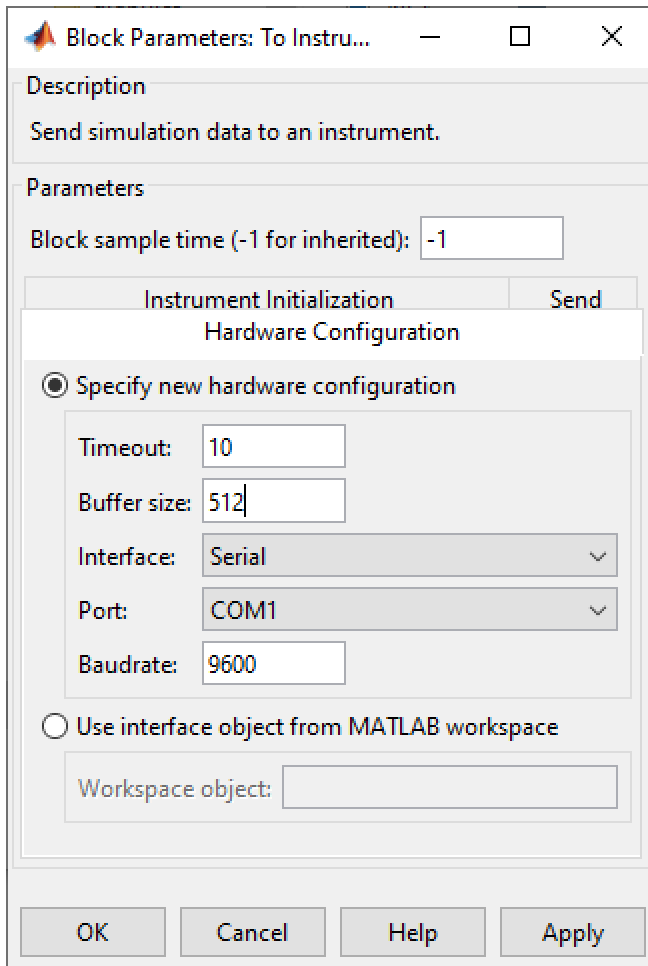


Note The two blocks do not directly connect within the model. The only communication between them is through the instrument, which is the loopback device connected to the serial port. Because the two blocks have no direct connection, you must consider their timing when running the model. The Query Instrument block does not get its input from the To Instrument block, so it has no way to know when the data from the instrument is available. Therefore, you must set the block parameters to write the data to the loopback before the model attempts to receive data from the loopback.

Step 6: Specify the Block Parameter Values

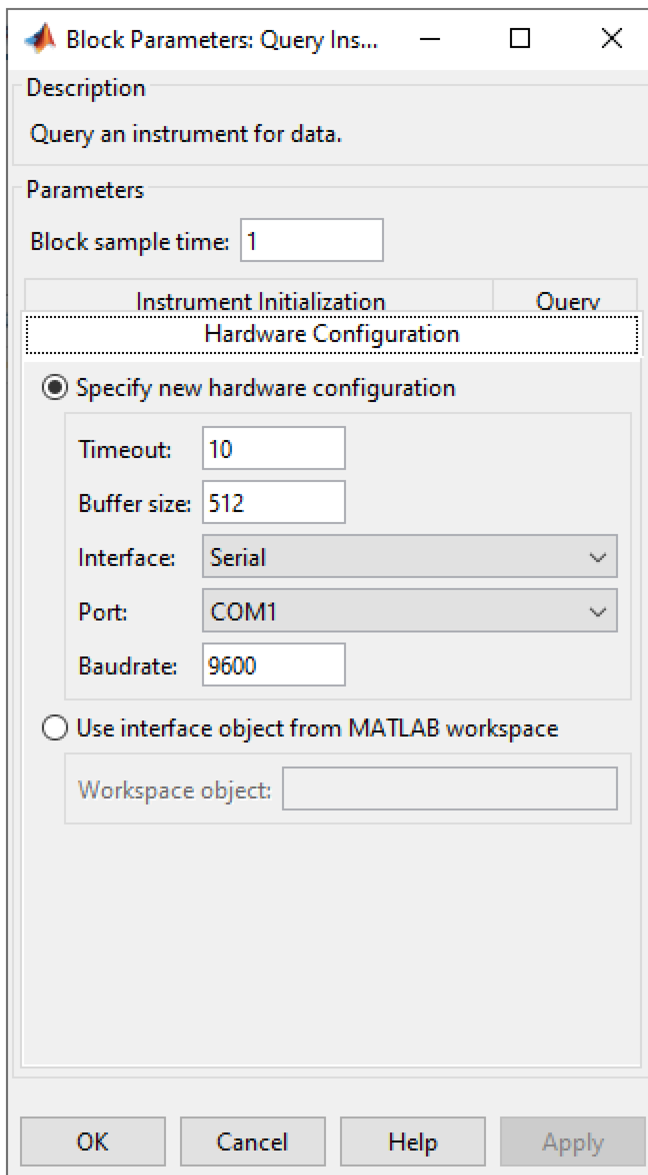
Set parameters for the blocks in your model by double-clicking the block.

Double-click the To Instrument block to open its parameters dialog box. Set **Port** to the port that your loopback device is connected to. For instructions, see “Find Serial Port Information for Your Platform” on page 6-9.



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

Double-click the Query Instrument block to open its parameters dialog box. Make sure that the values on the **Hardware Configuration** tab match the Hardware values on the To Instrument block.

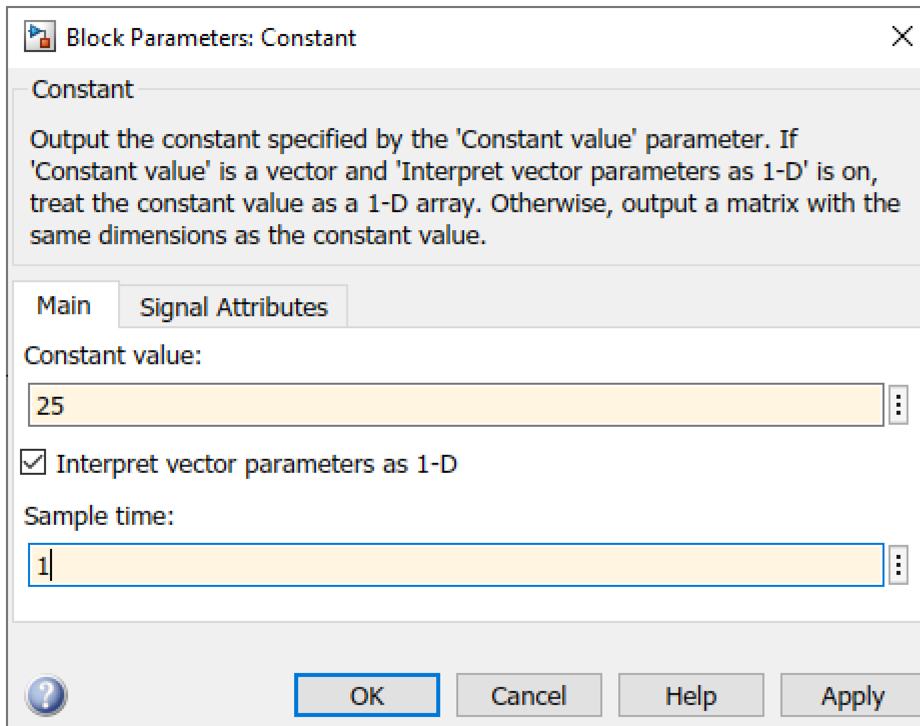


The model uses the default values on the **Instrument Initialization** and **Query** tabs of this block, so you do not need to modify any of their values.

Click **OK** to apply any changes and close the dialog box.

Double-click the Constant block to open its parameters dialog box. Change the Constant value to the value you want to send to the instrument. For this example:

- **Constant value** to 25.
- **Sample time** to 1.



Click **OK**.

For the Display block, you can use its default parameters.

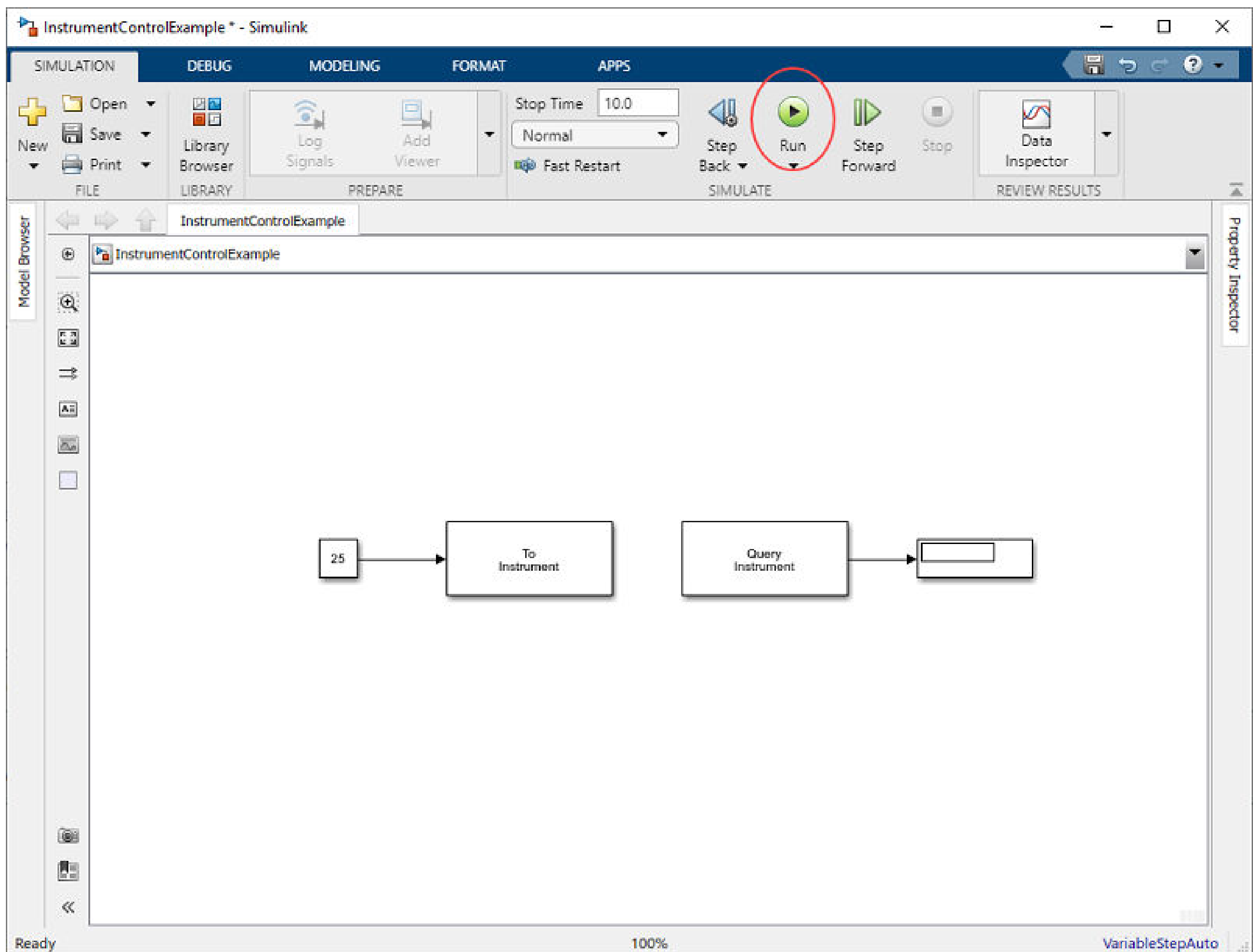
Step 7: Specify the Block Priority

The block with the lowest number gets the highest priority. In the Simulink Editor, right-click a block and select **Properties**. Enter the priority number in the **Priority** field in the Block Properties dialog box. To ensure that the To Instrument block first completes writing data to the loopback before the Query Instrument block reads it, set the priority of the To Instrument block to 1 and the Query Instrument block to 2.

Caution You must set the correct priority for the blocks in your model. Otherwise you might see unexpected results.

Step 8: Run the Simulation

To run the simulation, click the green **Run** button in the Simulink Editor toolstrip. You can use toolstrip options to specify how long to run the simulation and to stop a running simulation.



When you run the simulation, the constant value you specify (25) is written to the instrument (the serial loopback), received from the instrument, and shown in the Display block.

While the simulation is running, the status bar at the bottom of the Simulink Editor updates the progress of the simulation.

See Also

To Instrument | Query Instrument

More About

- “Send and Receive Data over TCP/IP Network” on page 23-14

Send and Receive Data over TCP/IP Network

This example shows how to build a simple model using the Instrument Control Toolbox blocks in the block library in conjunction with other blocks in the Simulink library. This example also illustrates how to send data to an echo server using TCP/IP and to read that data back into your model.

In this example, you create an echo server on your machine that simulates sending a signal to the TCP/IP Send block and echo the result back to the Send block to send data. You then use the TCP/IP Receive block to read that same data back into your model.

Note Block names are not shown by default in the model. To display the hidden block names while working in the model, select **Display** and clear the **Hide Automatic Names** check box.

In this section...
“Step 1: Create an Echo Server” on page 23-14
“Step 2: Create a New Model” on page 23-14
“Step 3: Open the Block Library” on page 23-15
“Step 4: Drag the Instrument Control Toolbox Blocks into the Model” on page 23-16
“Step 5: Drag the Sine Wave and Scope Blocks to Complete the Model” on page 23-16
“Step 6: Connect the Blocks” on page 23-18
“Step 7: Specify the Block Parameter Values” on page 23-19
“Step 8: Specify the Block Priorities” on page 23-21
“Step 9: Run the Simulation” on page 23-22
“Step 10: View the Result” on page 23-23

Step 1: Create an Echo Server

Open a port on your computer to work as an echo server that you can use to send and receive signals using TCP/IP. To create an echo server, run the following command in MATLAB.

```
echotcpip('on',50000)
```

Port 50000 opens on your machine to work as an echo server and turn it on.

Step 2: Create a New Model

To start Simulink and create a new model, enter the following at the MATLAB command prompt.

```
simulink
```

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

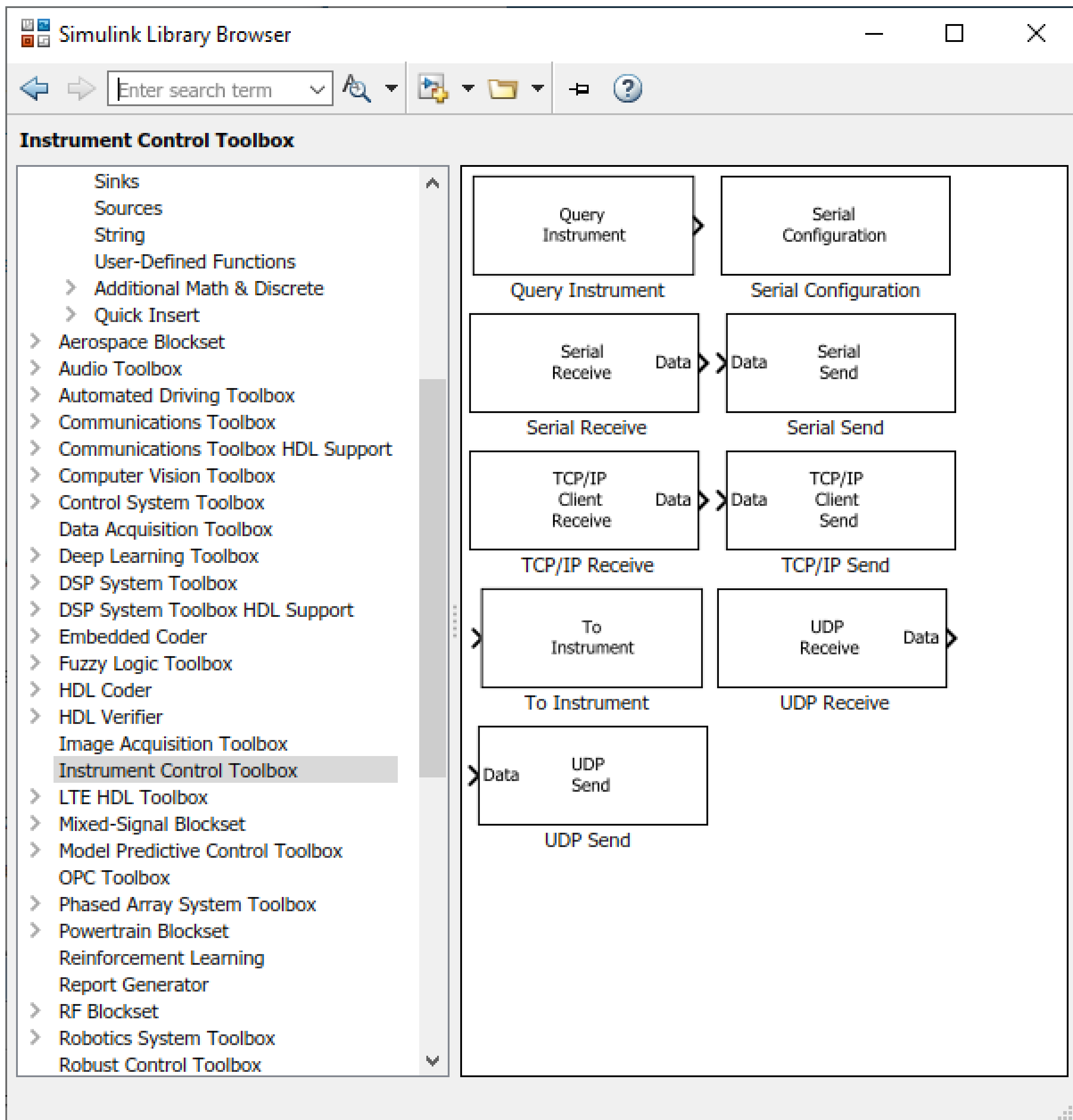
In the toolstrip, click **Save** to assign a name to your new model.

Step 3: Open the Block Library

In the toolstrip, click **Library Browser** in the **Simulation** tab.

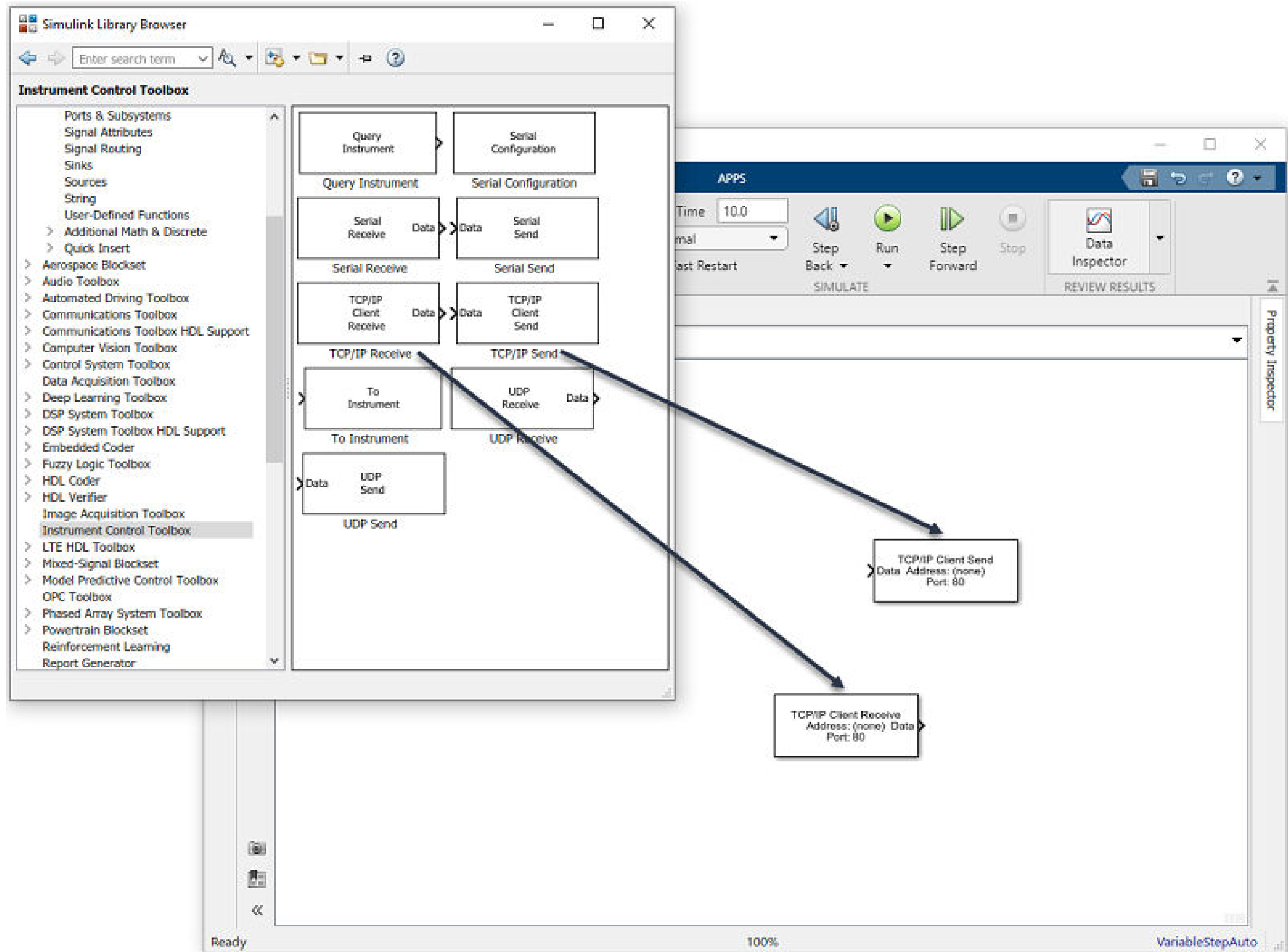
The Simulink Library Browser opens. The left pane contains a tree of available block libraries in alphabetical order. Click **Instrument Control Toolbox**.

To use a block, add it to an existing model or create a new model.



Step 4: Drag the Instrument Control Toolbox Blocks into the Model

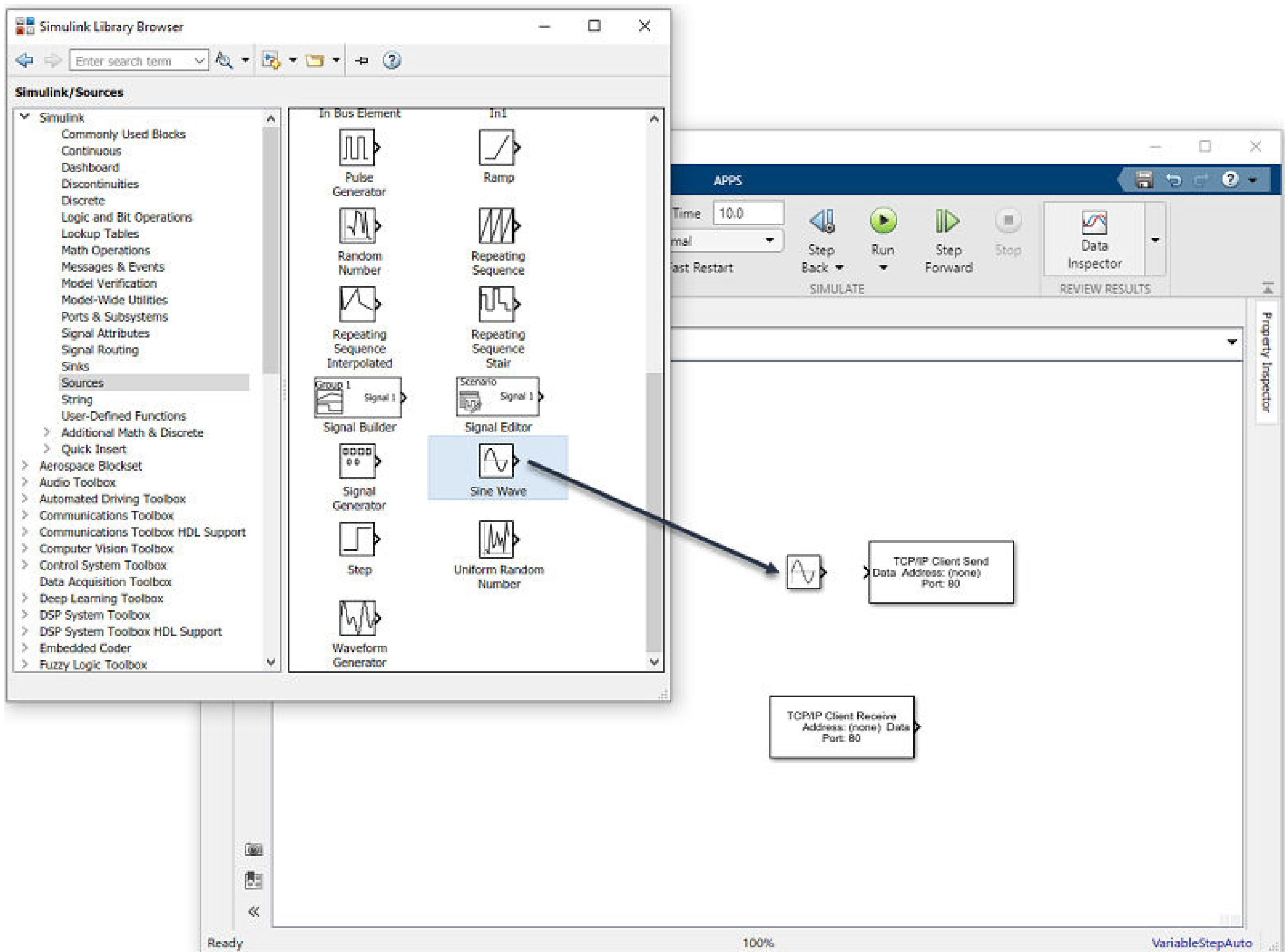
To use a block in a model, drag the block into the Simulink Editor. For this model, you need one instance of the TCP/IP Send and the TCP/IP Receive blocks in your model.



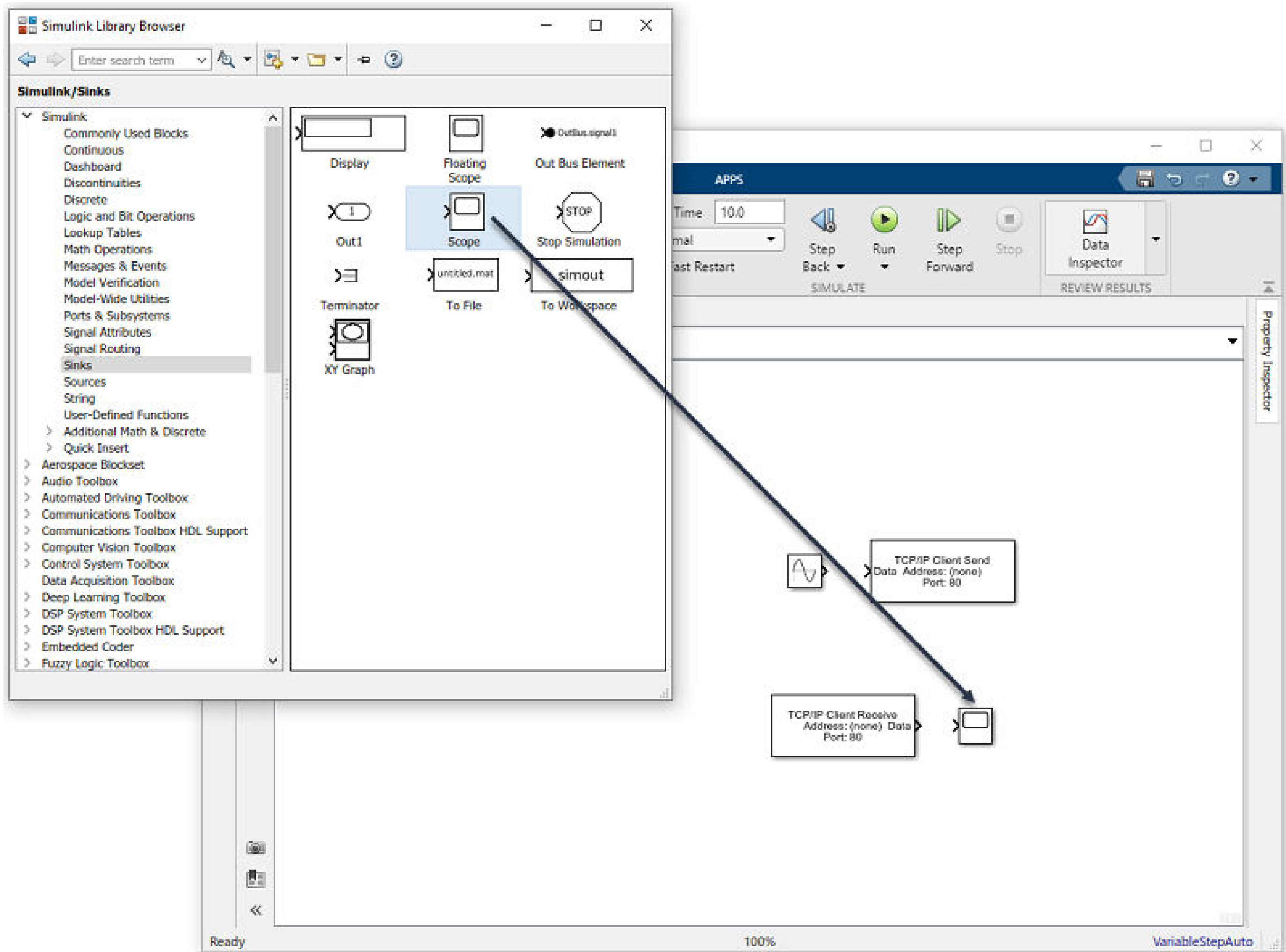
Step 5: Drag the Sine Wave and Scope Blocks to Complete the Model

This example requires two more blocks. One block displays the data received from the TCP/IP Receive block and the other block is the data to be sent to the TCP/IP Send block.

The TCP/IP Send block needs a data source for data to be sent. Add the Sine Wave block to the model to send signals to the TCP/IP Send block. To access the Sine Wave block, expand the **Simulink** node in the browser tree, and click the **Sources** library entry. From the blocks in the right pane, drag the Sine Wave block into the model and place it to the left of the TCP/IP Send block.

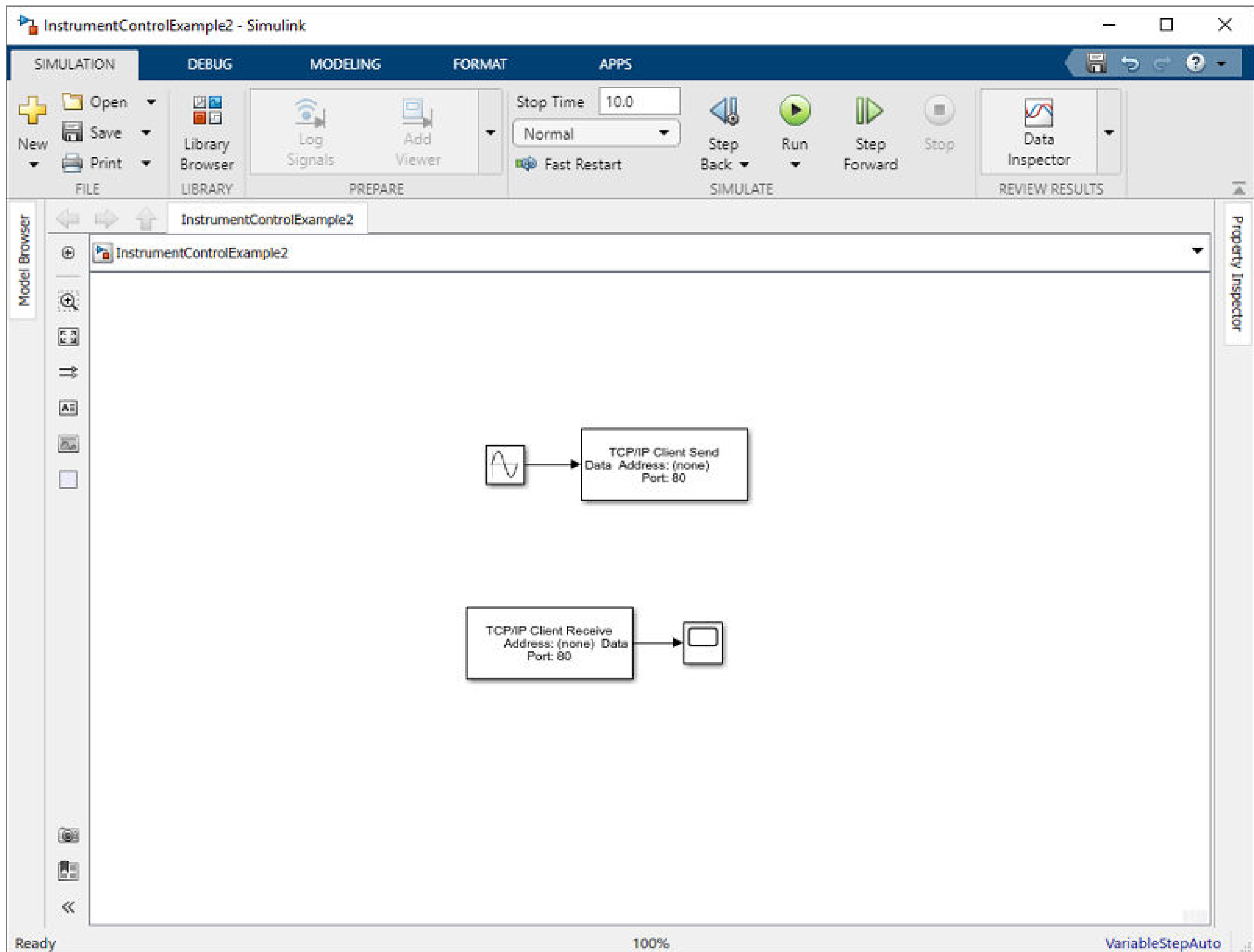


To display the data received by the TCP/IP Receive block, use the Scope block. To access this block, click the **Sinks** library entry in the expanded **Simulink** node in the browser tree. From the blocks in the right pane, drag the Display block into the model and place it to the right of the TCP/IP Receive block.



Step 6: Connect the Blocks

Make a connection between the Sine Wave block and the TCP/IP Send block. A quick way to make the connection is to select the Sine Wave block, press and hold the **Ctrl** key, and then click the TCP/IP Send block. In the same way, make the connection between the output port of the TCP/IP Receive block and the input port of the Scope block.

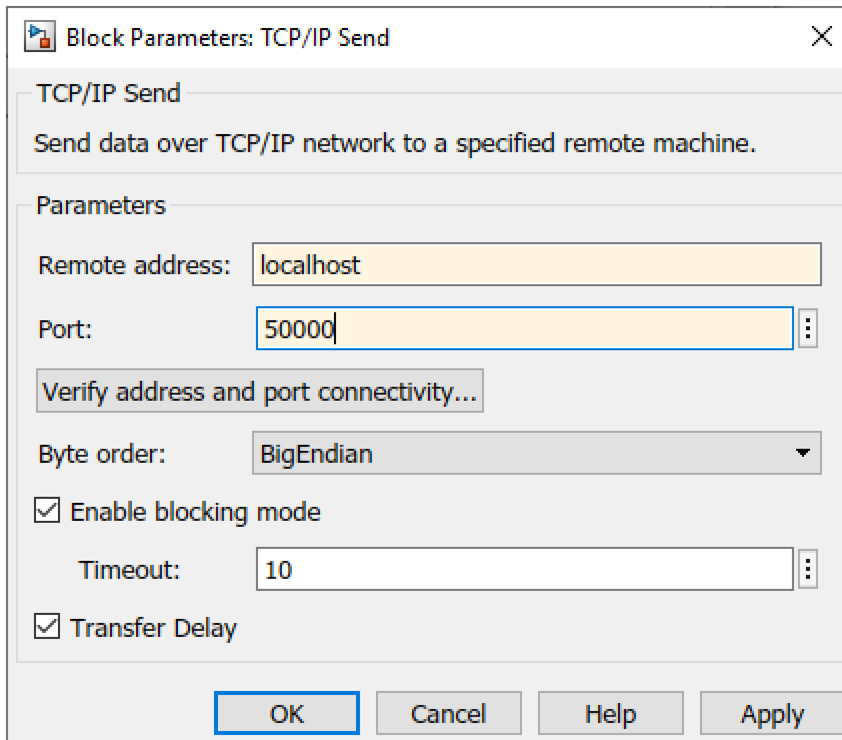


Step 7: Specify the Block Parameter Values

Set parameters for the blocks in your model by double-clicking the block.

Configure the Send Block

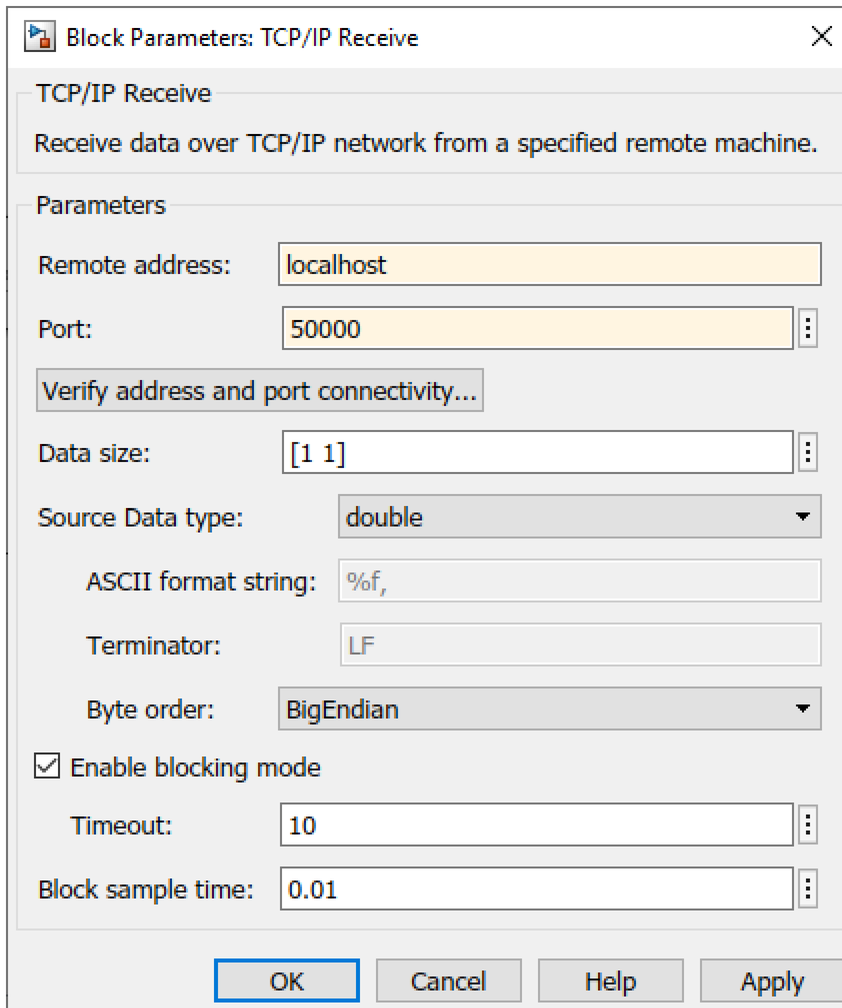
Double-click the TCP/IP Send block to open its parameters dialog box. Set the **Remote address** field to `localhost` and the **Port** field to `50000`, since that is the address you set the echo server to when you started it.



Click **Apply** and then **OK**.

Configure the Receive Block

Double-click the Receive block to open its parameters dialog box. Set the **Remote address** field to `localhost` and the **Port** field to `50000`. Change the **Data type** to `double`. The **Block sample time** field is set to `0.01` by default. The block sample time here must match the one in the Sine Wave block, so confirm that they are both set to `0.01`.



Click **OK**.

Configure the Sine Wave Block

Double-click the Sine Wave block to open its parameters dialog box. Set the **Sample time** field to `0.01`.

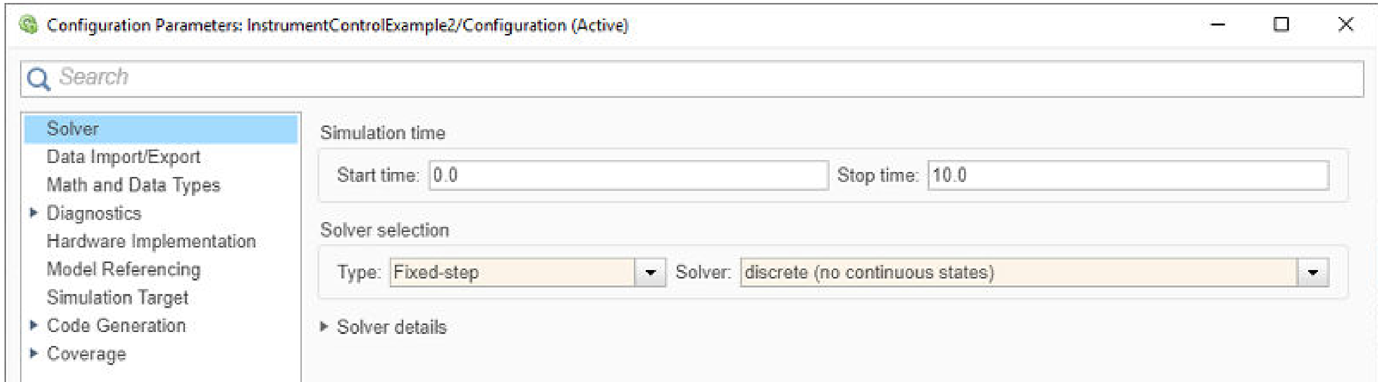
Click **OK**.

Step 8: Specify the Block Priorities

To run the simulation correctly, specify the order in which Simulink processes the blocks. Right-click the block and select **Properties**. Enter the priority number in the **Priority** field. For this example, set the priority of TCP/IP Send to 1 and TCP/IP Receive to 2.

Caution You must set the correct priority for the blocks in your model. Otherwise you might see unexpected results.

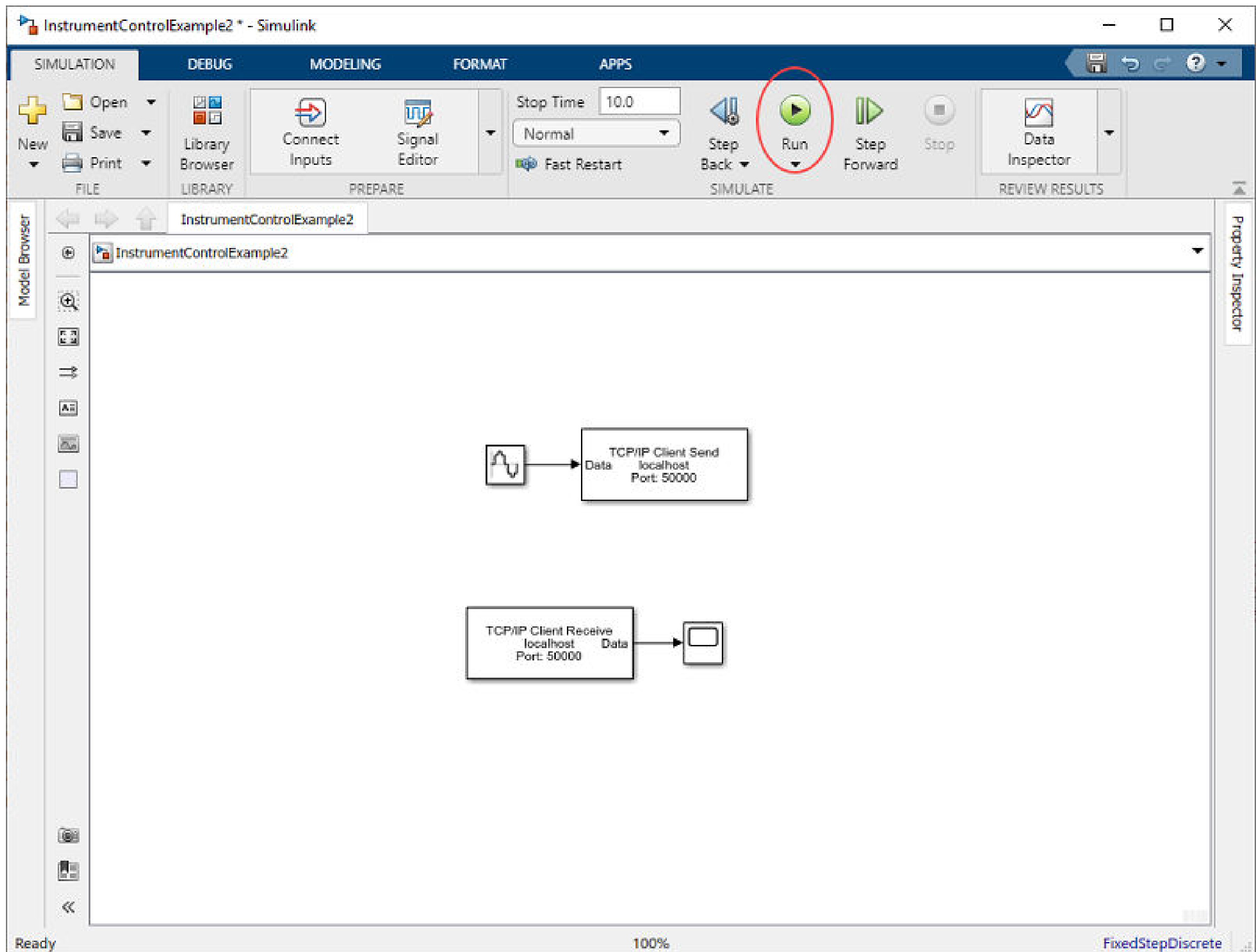
You also need to set two parameters on the model. In the Simulink toolstrip, click **Model Settings** from the **Prepare** section on the **Simulation** tab. In the Configuration Parameters dialog box, set the **Type** field to `Fixed-step` and set the **Solver** field to `discrete (no continuous states)`.



Click **OK**.

Step 9: Run the Simulation

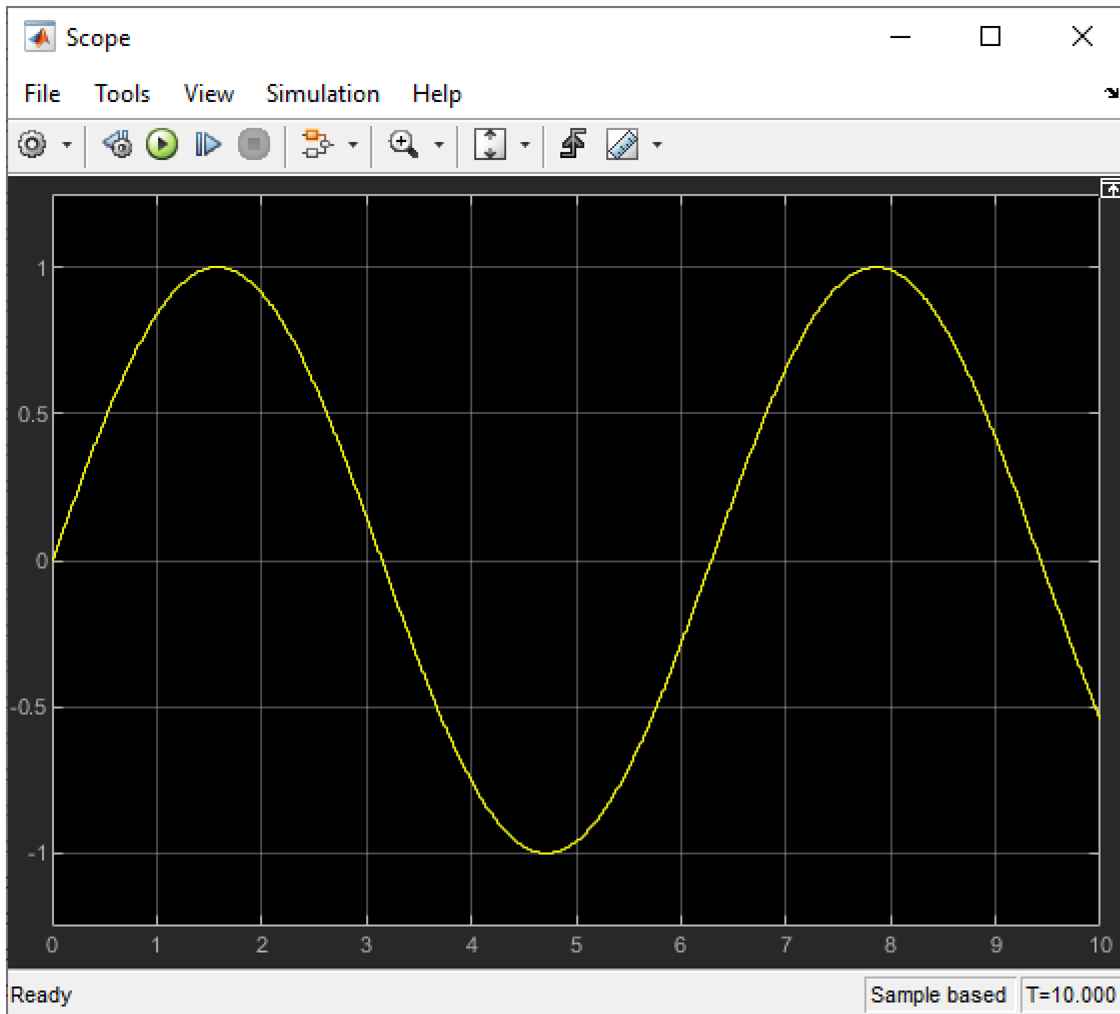
To run the simulation, click the green **Run** button on the Simulink Editor toolstrip. You can use toolstrip options to specify how long to run the simulation and to stop a running simulation.



While the simulation runs, the status bar at the bottom of the Simulink Editor updates the progress of the simulation.

Step 10: View the Result

Double-click the Scope block to view the signal on a graph as it is received by the TCP/IP Receive block.



For more information about Instrument Control Toolbox blocks, see the blocks reference documentation.

See Also

TCP/IP Receive | TCP/IP Send

More About

- “Send and Receive Data Through Serial Port Loopback” on page 23-5

Enable Blocking Mode in Receive and Send Blocks

The **Enable blocking mode** parameter allows you to wait until the requested data is available for the block to receive or wait until all the data sends. You can disable this option to allow your simulation to run continuously. This parameter is available in the following blocks:

- Serial Receive
- Serial Send
- TCP/IP Receive
- TCP/IP Send
- UDP Receive
- UDP Send

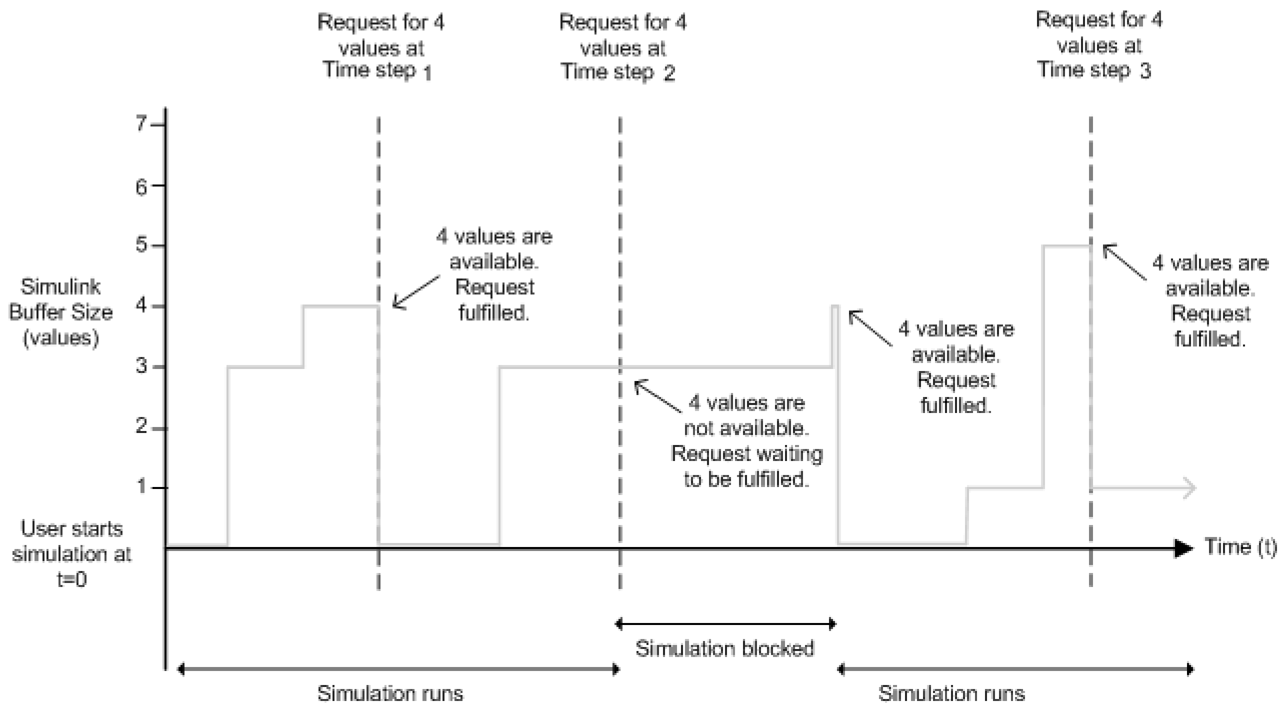
Blocking Mode

The **Enable blocking mode** parameter is on by default.

Receive Blocks

If you enable blocking mode in the Serial Receive, TCP/IP Receive, and UDP Receive blocks, the simulation waits for the requested data to become available. The model waits for up to the amount of time specified by the **Timeout** parameter.

In this example, start the simulation at time t = 0 and specify the **Data size** as `[4,1]`. Once the simulation starts, the data is acquired asynchronously in a FIFO buffer.



The blocking mode simulation occurs in the following steps.

- At time step 1: The Simulink software requests data and the buffer size is four values. The block fulfills the request without interrupting the simulation. The block resets the buffer size value to 0.
- At time step 2: The Simulink software requests data again, and the buffer size is only three values; therefore, the software blocks the simulation until it receives the fourth value. When the block receives the fourth value, it fulfills the request and resumes the simulation. The block resets the buffer size value to 0.
- At time step 3: When Simulink software requests data, the block has five values. It returns the first four that it received and resets the buffer size value to 1.

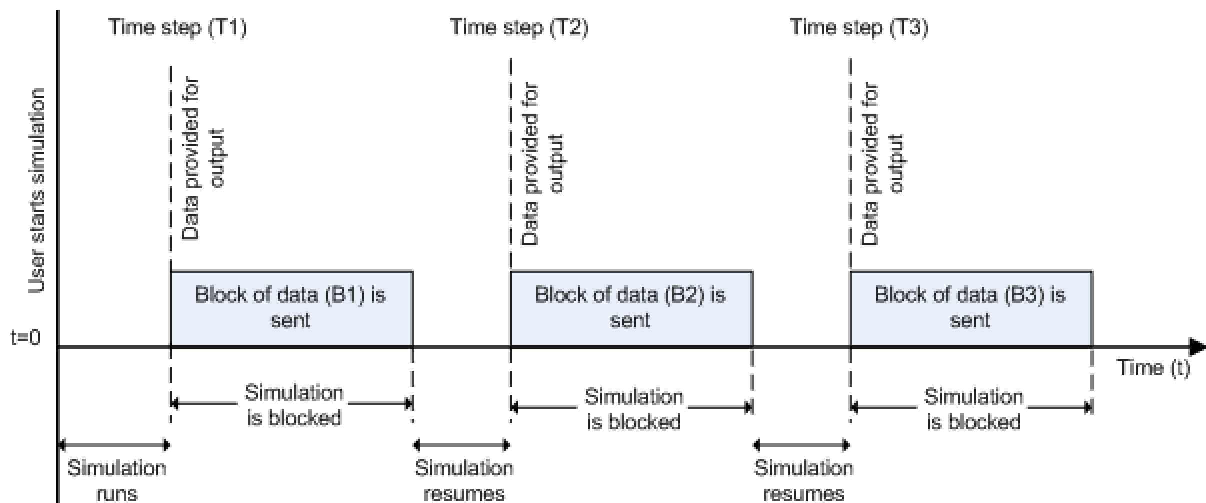
For each time step, if the requested data is not received within the amount of time specified in the **Timeout** field:

- The Serial Receive block takes the action specified by the **Action when data is not available** parameter.
- The TCP/IP Receive block outputs a value of `0`.

Send Blocks

If you enable blocking mode in the Serial Send, TCP/IP Send, and UDP Send blocks, the simulation waits until the block sends complete data.

In this example, start the simulation at time t=0.



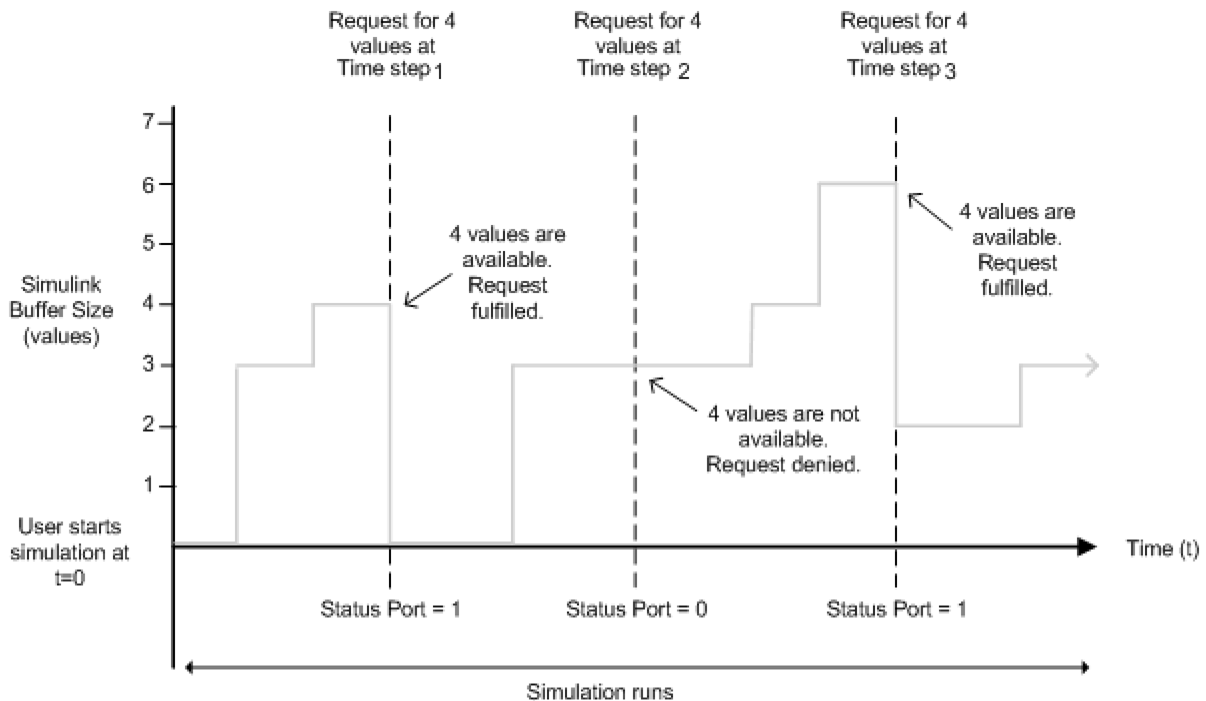
At time step (T1), data output is initiated and simulation stops until the block of data (B1) is sent. After the data is sent, simulation resumes until time step (T2), where the block initiates another data output and simulation is blocked until the block of data (B2) is sent, and the simulation resumes.

Nonblocking Mode

Unselect the check box if you want the **Enable blocking mode** parameter to be off.

Receive Blocks

If you do not enable blocking mode in the Serial Receive, TCP/IP Receive, and UDP Receive blocks, the simulation runs continuously and the block has two output ports, **Status** and **Data**. The **Data** port contains the requested set of data at each time step. The **Status** port contains 0 or 1 based on whether it received new data at the given time step.



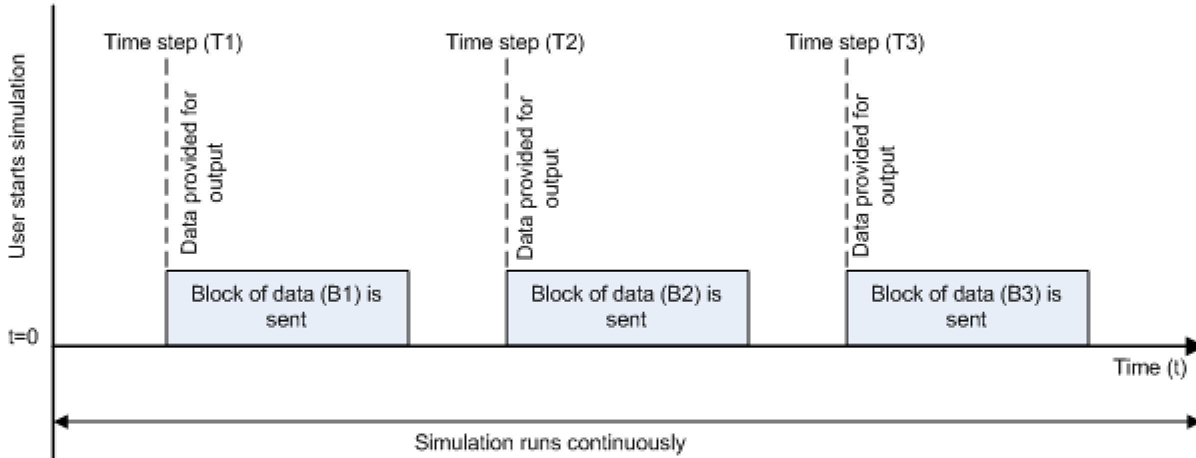
Here the simulation is not blocked and runs continuously.

- At time step 1: The Simulink software requests data and the buffer size is four values. The block fulfills the request and changes the **Status** port value to `1`, indicating that new data is available. The **Data** port at this point contains the newly received values. The block resets the buffer size value to 0.
- At time step 2: The Simulink software requests data again, and the buffer size is only three values. The block cannot return a value of 3 because the data size is specified as 4. Therefore, the block sets the **Status** port value to `0`, indicating that no new data is available. The **Data** port contains the previously received value or 0, depending on the block, and the buffer size is at three (the number of values it received since the last request was fulfilled).
- At time step 3: When the Simulink software requests data here, the buffer size is now six values. The block returns the first four in the order received and changes the **Status** port value to `1`.

Send Blocks

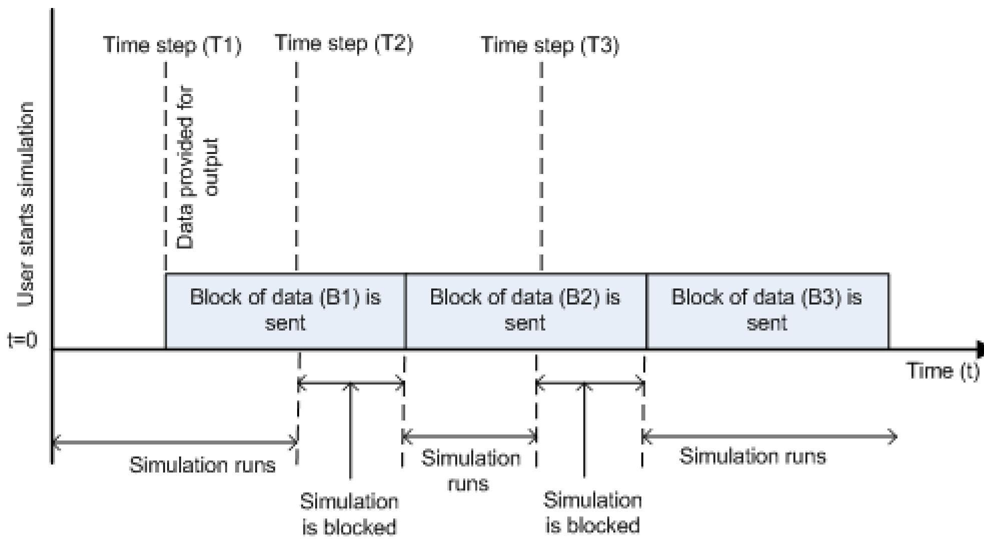
If you do not enable blocking mode in the Serial Send, TCP/IP Send, and UDP Send blocks, the simulation runs continuously.

Data output outpaces simulation speed



In this scenario, the data output outpaces the simulation speed. Data output is initiated at the first time step (T1) and the corresponding block of data (B1) is sent to the specified remote address asynchronously. The simulation runs continuously in this mode.

Simulation speed outpaces data output



In this scenario, the simulation speed outpaces the data acquisition.

- At time step T1: The block of data (B1) is sent asynchronously.
- At time step T2: The simulation is blocked until the block of data (B1) is sent completely. When B1 is completely sent, the new block of data (B2) is sent asynchronously, and the simulation resumes.

Note Several factors, including network connectivity and model complexity, can affect the simulation speed, which can cause both nonblocking scenarios to occur within the same simulation.

See Also

Serial Receive | Serial Send | TCP/IP Receive | TCP/IP Send | UDP Receive | UDP Send

Timing in Hardware Interface Models

Simulation Time

When blocks in your Simulink model must interface with hardware devices, you might have to consider how long the simulation takes to run in real time versus simulation time, and how often and how many times the hardware interface blocks execute during a simulation. Usually your hardware communication rates are relative to real-world or "wall clock" time. You can adjust the duration of a simulation, the execution rate of the blocks, and the pacing of the model to accommodate your hardware requirements. This topic discusses basic timing concepts in hardware interface models, using fixed steps for block execution.

A model simulation has a duration defined by a start time and a stop time. The default duration is 10 units of simulation time (or simulated seconds). These simulation seconds are not necessarily equivalent to a real-time second as measured by a wall clock.

To adjust the model duration, open the model Configuration Parameters by clicking the **Model Settings** icon in the Modeling tab of the model editor toolstrip. Select **Solver** in the left pane. The **Start time** and **Stop time** settings define the duration. In most cases, **Start time** should be `0.0`, and you can set **Stop time** to reflect the duration you want the model to have.

As a simulation runs, the clocking for block execution is performed by a series of timesteps. With a setting for an automatic solver with fixed timestep sizes, during compilation Simulink calculates the timestep frequency to accommodate the **Sample time** parameter settings of all the blocks in the model. For example, if all the timed blocks in the model have a Sample time setting of 0.01 or a multiple of that, then a timestep size of 0.01 works for the whole model.

Block Sample Time

For models that interface with hardware devices, you might prefer fixed timesteps of a specified rate. For example, you might need millisecond resolution to control the timing relationship of your blocks. Set the timing options as follows:

- **Start time**: `0.0`
- **Stop time**: `10.0`
- **Type**: `Fixed-step`
- **Solver**: `discrete`
- **Fixed-step size**: `0.001`

The dialog settings look like this figure:

In this model, a block with a default **Sample time** setting of `0.01` executes every tenth timestep, or 1001 times in a 10 second simulation. Another block that must run at twice the rate should have **Sample time** set to `0.005`.

Note In most cases, you can leave the **Fixed-step size** setting to `auto`, allowing Simulink to calculate the appropriate fundamental sample time based on all the block settings.

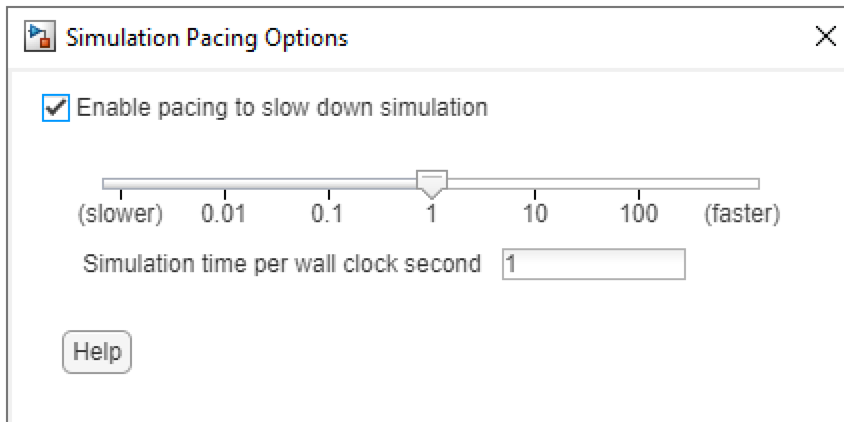
Because the simulation duration is 10 simulated seconds, and the **Sample time** period of the block is 0.01 simulated seconds, that block executes 1001 times in a complete simulation (including first and last step). The simulation runs as fast as its blocks can perform, and those 1001 executions might take significantly less than 10 seconds of wall clock time. So the simulation in real time is determined by how fast it can execute the blocks in the model for the required number of iterations. Often the purpose of simulation is to model behavior in a way that takes less time than it would in a real-world situation. In these cases, the sequence and repetition of block execution is important, while the actual span of real-world time might not be.

Pacing Model Simulation

You might have a requirement for a model to interact with a hardware device by repeating some operation at fixed intervals of real-world time. For example, a block might repeatedly read data from a thermometer or send triggers for an external signal generator to output a pulse train.

If you set the block **Sample time** to `0.1`, that would control the rate of block execution only in simulation time. To correlate simulation time to real time, you can use Simulation Pacing to slow down a simulation to run at the pace of real-world time. Access the Simulation Pacing Options dialog by clicking **Run** > **Simulation Pacing** in the Simulation tab of the model editor toolstrip

Check **Enable pacing to slow down simulation**, and set the slider ratio to `1` (the default). This causes simulation time to track as closely as possible with wall clock time, so 1 simulation second is approximately equal to 1 wall clock second.



With this pacing setting, a block **Sample time** of `0.1` is approximately equal to 0.1 wall clock seconds, resulting in ten block executions per second. So a block that generates a device output pulse every 0.1 simulation seconds, now puts out 10 pulses per wall clock second.

See Also

More About

- "What Is Sample Time?" (Simulink)
- "Simulation Pacing" (Simulink)

24

Functions

Test and Measurement Tool

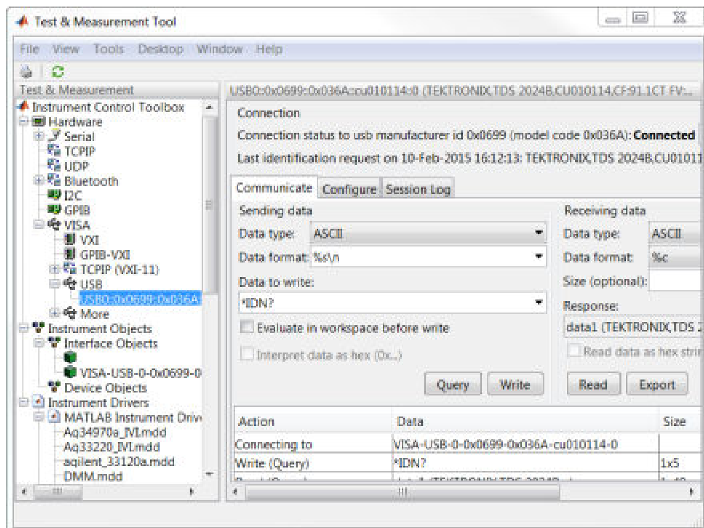
Control oscilloscopes and other instruments

Description

The **Test & Measurement Tool** enables you to configure and control resources, such as instruments, serial devices, drivers, and interfaces, through Instrument Control Toolbox without having to write a MATLAB script.

Use the Test & Measurement Tool to manage your session with the toolbox. You can:

- Detect available hardware and drivers.
- Connect to an instrument or device.
- Configure instrument or device settings.
- Read and write data.
- Automatically generate the MATLAB script.
- Visualize acquired data.
- Export acquired data to the MATLAB workspace.



Open the Test and Measurement Tool App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app icon.
- MATLAB command prompt: Enter `tmtool`.

Examples

- "Using the Test & Measurement Tool" on page 18-4

Programmatic Use

`tmtool` opens the Test & Measurement Tool, which enables you to configure and control resources, such as instruments, serial devices, drivers, and interfaces, accessible through the Instrument Control Toolbox.

See Also

Topics
"Using the Test & Measurement Tool" on page 18-4

Introduced before R2006a

Modbus Explorer

Read and write to Modbus coils and registers

Description

The **Modbus Explorer** app enables you to read and write to registers through Instrument Control Toolbox without having to write a MATLAB script.

The **Modbus Explorer** app offers a user interface to easily set up read and write operations, and a live plot to see the values. The read table allows you to easily organize and manage reads for multiple addresses, such as different sensors and switches on a PLC. The app supports a subset of the MATLAB MATLAB functionality. You can do the following in the **Modbus Explorer** app:

- Read coils, inputs, input registers, and holding registers. This is the functionality of the Modbus `read` function.
- Write to coils and holding registers. This is the functionality of the Modbus `write` function.

The app does not support the functionality of the Modbus `writeRead` function or the `maskWrite` function.

Open the Modbus Explorer App

- MATLAB Toolstrip: On the **Apps** tab, under **Test & Measurement**, click the app icon.
- MATLAB command prompt: Enter `modbusExplorer`.

Examples

- “Configure a Connection in the Modbus Explorer” on page 11-20
- “Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23
- “Write to Coils and Holding Registers in the Modbus Explorer” on page 11-26
- “Control a PLC Using the Modbus Explorer” on page 11-28

See Also

Topics
“Configure a Connection in the Modbus Explorer” on page 11-20
“Read Coils, Inputs, and Registers in the Modbus Explorer” on page 11-23
“Write to Coils and Holding Registers in the Modbus Explorer” on page 11-26
“Control a PLC Using the Modbus Explorer” on page 11-28

Introduced in R2019a

Serial Explorer

Communicate with devices connected to serial port

Description

The **Serial Explorer** app creates a connection to a serial port on your machine. After you connect to a serial port, you can communicate with it, plot and analyze data, export data to the workspace, and generate MATLAB code.

Using this app, you can:

- Configure serial port communication properties.
- Send binary or string data to the connected serial port.
- Read binary or string data sent from the connected serial port.
- Plot data in a figure window.
- Analyze data by viewing it in the **Signal Analyzer** app.
- Export data to the workspace.
- Generate a MATLAB Live Script file that uses the `serialport` interface.

Open the Serial Explorer App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app icon.
- MATLAB command prompt: Enter `serialExplorer`.

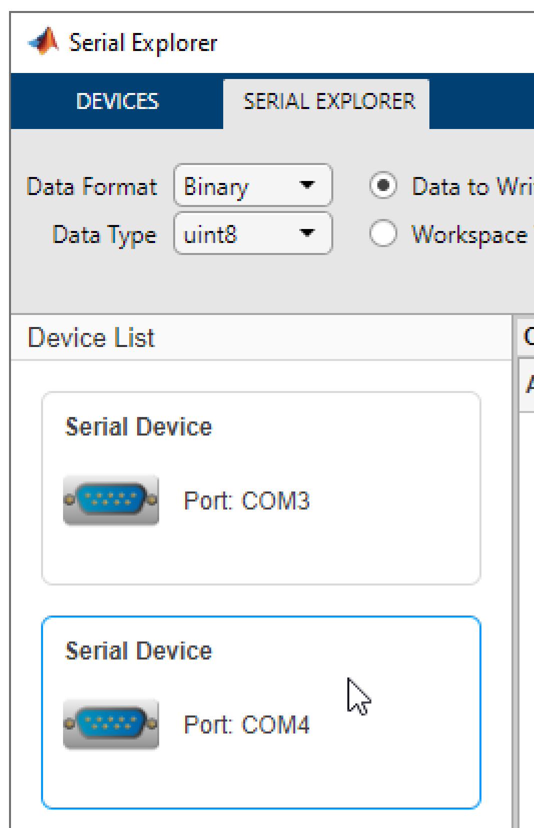
Examples

Write ASCII-Terminated Command to Serial Port and Read Response

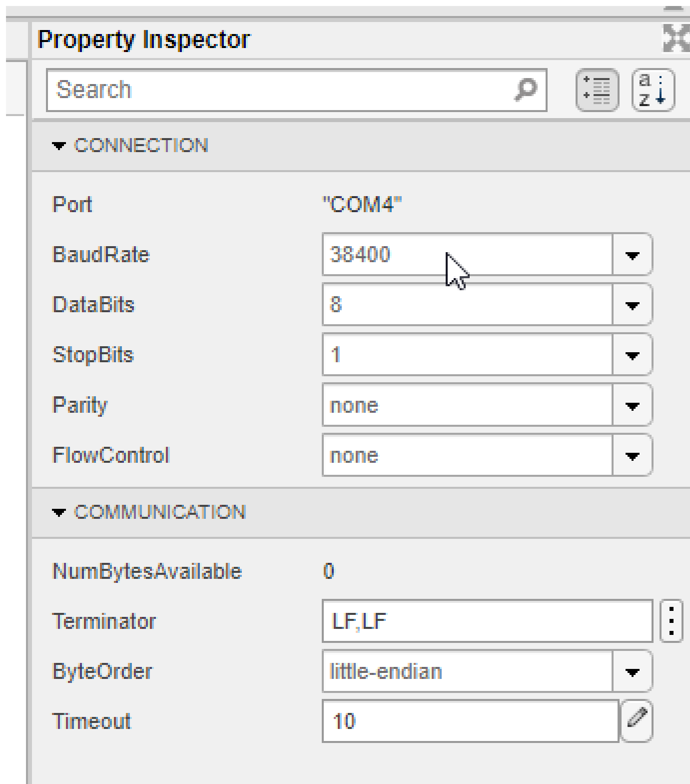
In this example, write ASCII-terminated data to a serial port device and read data back from it. The device in this example is an Arduino® Uno that has already been programmed with custom commands and responses.

Open the **Serial Explorer** app either from the **Apps** tab in the MATLAB toolstrip or the MATLAB command prompt.

In the **Device List** pane, select the serial port device that you want to communicate with. In this example, the device is connected to COM4.

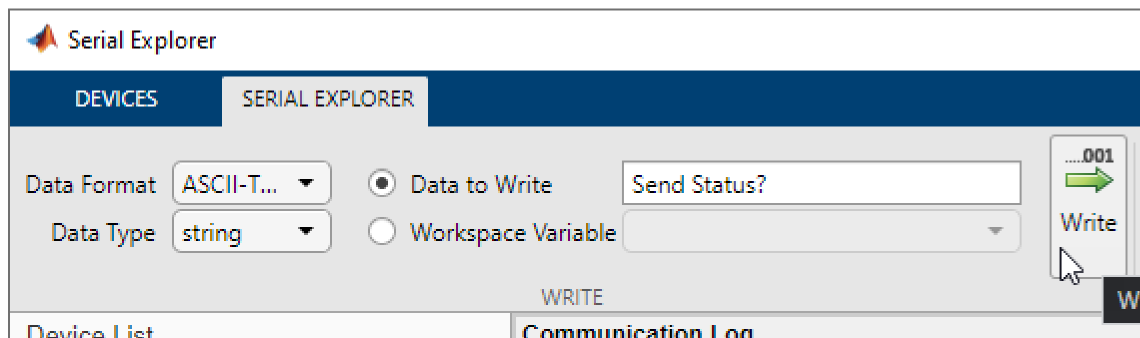


Configure **Connection** and **Communication** properties from the **Property Inspector**. Specify these device properties before writing and reading data to match the appropriate value for the connected device. For this example, change the **BaudRate** to `38400`.

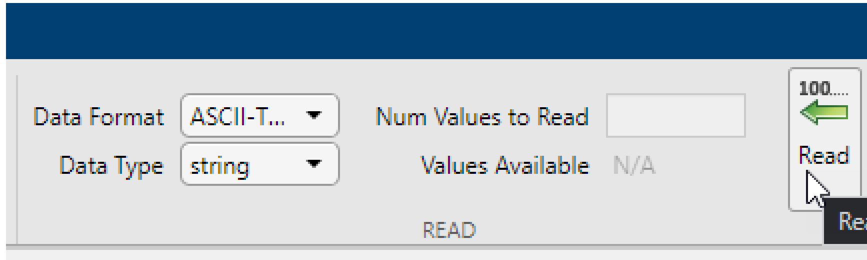


Some serial port devices can accept string queries and respond to them. In this example, the connected Arduino Uno has been programmed to receive and respond to customized string commands. The commands in this example do not work for other devices.

Send the `Send Status?` command to the device. In the **Write** section, set the **Data Format** to `ASCII-Terminated String`. The **Data Type** changes to `string` since that is the only possible option. Specify the **Data to Write** as `Send Status?`. Click **Write** to write the data to the serial port device. For ASCII-terminated string write operations, the write terminator specified by the **Terminator** property is automatically appended to the data being written.



You can view the response to this command by reading from the serial port device. In the **Read** section, set the **Data Format** to `ASCII-Terminated String`, which changes the **Data Type** to `string`. Read the data from the device by clicking **Read**. Data is read until the first occurrence of a terminator.



View both the write and read operations in the **Communication Log** pane. The read operation shows the message `Arduino connected to 'COM4'`. You can select a row to export it as a variable to the workspace by following the steps in “Export Data from Communication Log and Generate MATLAB Script” on page 24-17.

Communication Log

Action	Data	Size	Data Type	Time
WriteLine	Send Status?	1 x 1	string	22-Jun-2021 11:44:51
ReadLine	Arduino connected to 'COM4'	1 x 1	string	22-Jun-2021 11:51:59

The **MATLAB Code Log** pane shows the code for these operations. You can export this code as a MATLAB Live Script file by following the steps in “Export Data from Communication Log and Generate MATLAB Script” on page 24-17.

MATLAB Code Log

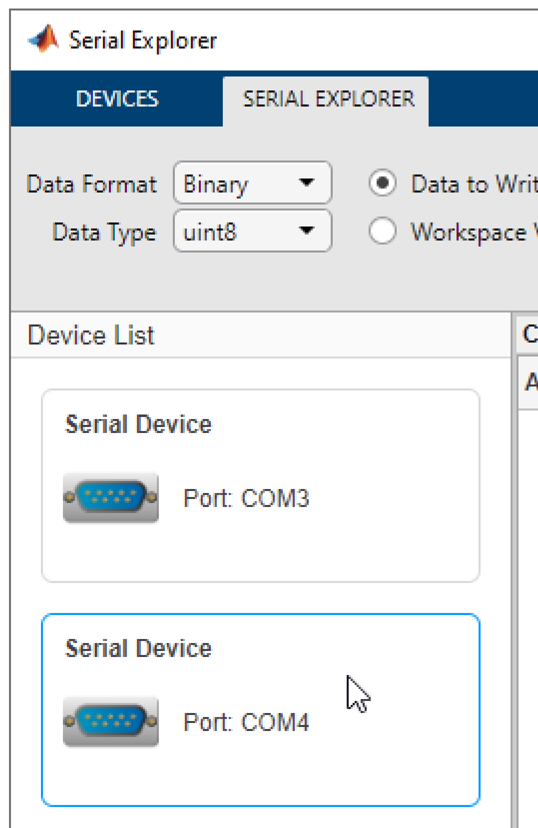
```
% Create a serialport object called serialportObj that connects to the port "COM4" with a
% default baud rate of 9600.
serialportObj = serialport("COM4",9600);

% Set the value of the "BaudRate" property of the serialport object serialportObj to
% 38400.
serialportObj.BaudRate = 38400;

% Write the data "Send Status?" as a string using the serialport object serialportObj.
% The write terminator "LF" is automatically appended to the data before writing.
writeline(serialportObj,"Send Status?");

% Read string data up to and including the first occurrence of the read terminator "LF"
% using the serialport object serialportObj. Data is returned without the read
% terminator.
data1 = readline(serialportObj);
```

Read Binary Data from Serial Port and Plot It

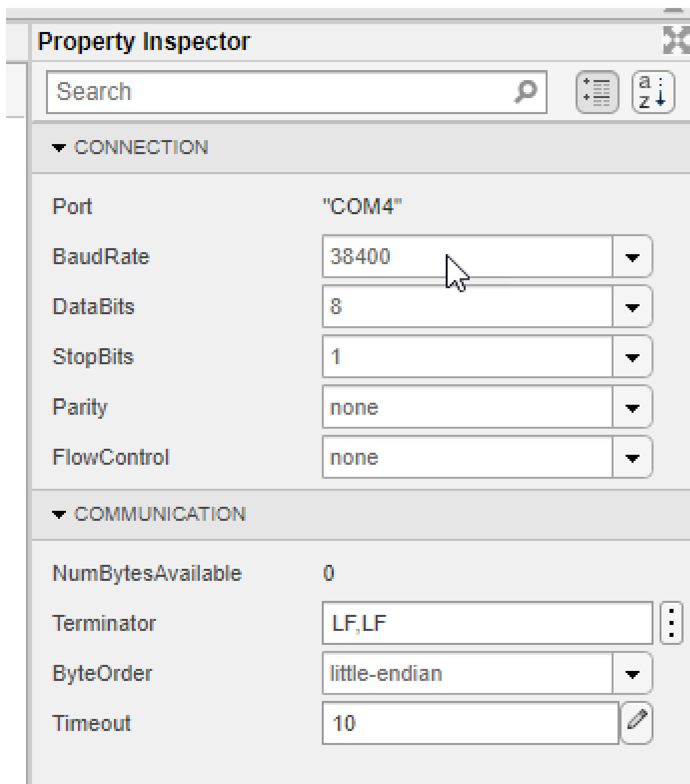
In this example, write ASCII-terminated data to a serial port device and read data back from it. The device in this example is an Arduino Uno that has already been programmed with custom commands and responses.

Open the **Serial Explorer** app from either the **Apps** tab in the MATLAB toolstrip or the MATLAB command prompt.

In the **Device List** pane, select the serial port device that you want to communicate with. In this example, the device is connected to COM4.

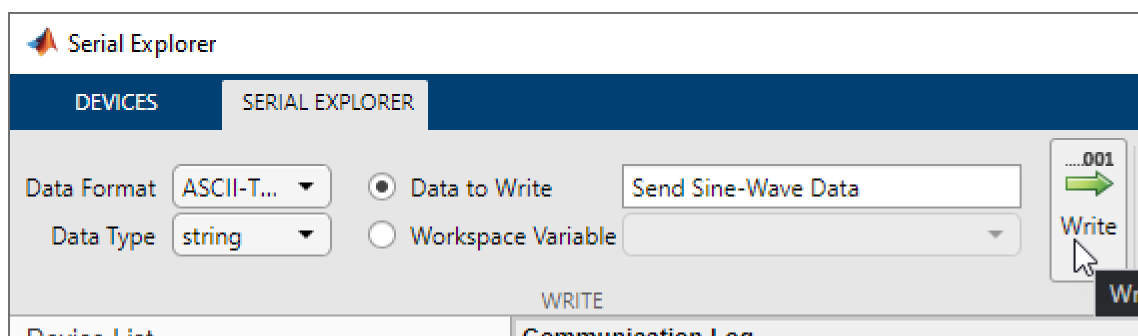


Configure **Connection** and **Communication** properties from the **Property Inspector**. Specify these device properties before writing and reading data to match the appropriate value for the connected device. For this example, change the **BaudRate** to `38400`.



Some serial port devices can accept string queries and respond to them. In this example, the connected Arduino Uno has been programmed to receive and respond to customized string commands. The commands in this example do not work for other devices.

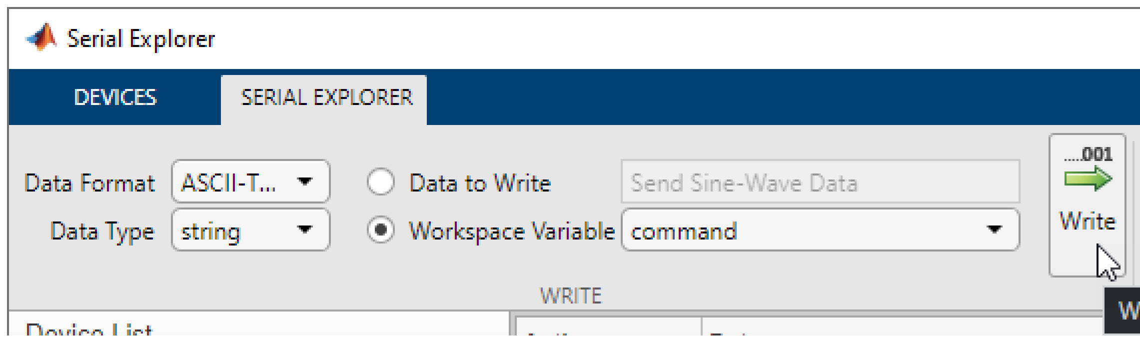
Send the `Send Sine-Wave Data` command to the device. In the **Write** section, set the **Data Format** to `ASCII-Terminated String`. The **Data Type** changes to `string` since that is the only possible option. Specify the **Data to Write** as `Send Sine-Wave Data`. Click **Write** to write the data to the serial port device. For ASCII-terminated string write operations, the write terminator specified by the **Terminator** property is automatically appended to the data being written.



Send another command to the device. In the MATLAB command prompt, create a workspace variable for this command.

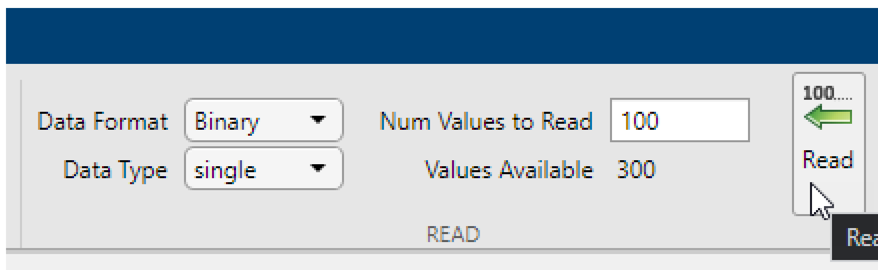
```
command = "Send Arbitrary Waveform";
```

Select **Workspace Variable** and select the `command` option. Click **Write**.

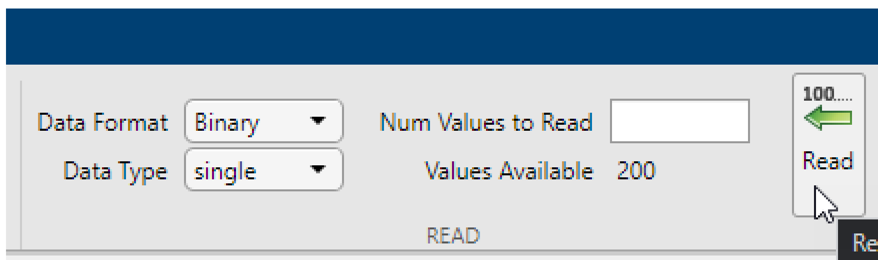


Before reading data from the serial port device, you must specify the correct data format and type. In this example, the responses to the string commands are stored as binary data with single precision. In the **Read** section, set the **Data Format** to `Binary`, and the **Data Type** to `single`. The **Values Available** parameter is `300`. The first 100 values are the response to the `Send Sine-Wave Data` command and the remaining 200 values are the response to the `Send Arbitrary Waveform` command.

Specify the **Num Values to Read** as `100`. Read the first 100 values of the data from the serial port device by clicking **Read**.



If you do not specify a value for the **Num Values to Read** parameter, you can read all the available values. Read the remaining 200 values by clearing the **Num Values to Read** parameter and clicking **Read**.



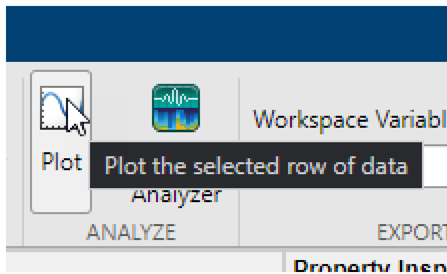
View both the write and read operations in the **Communication Log** pane. You can select a row to plot it, view it in the **Signal Analyzer** app, or export it as a variable to the workspace. Select the data from the first read operation.

WRITE READ COMMUNICATION LOG ANALYZE EXPORT

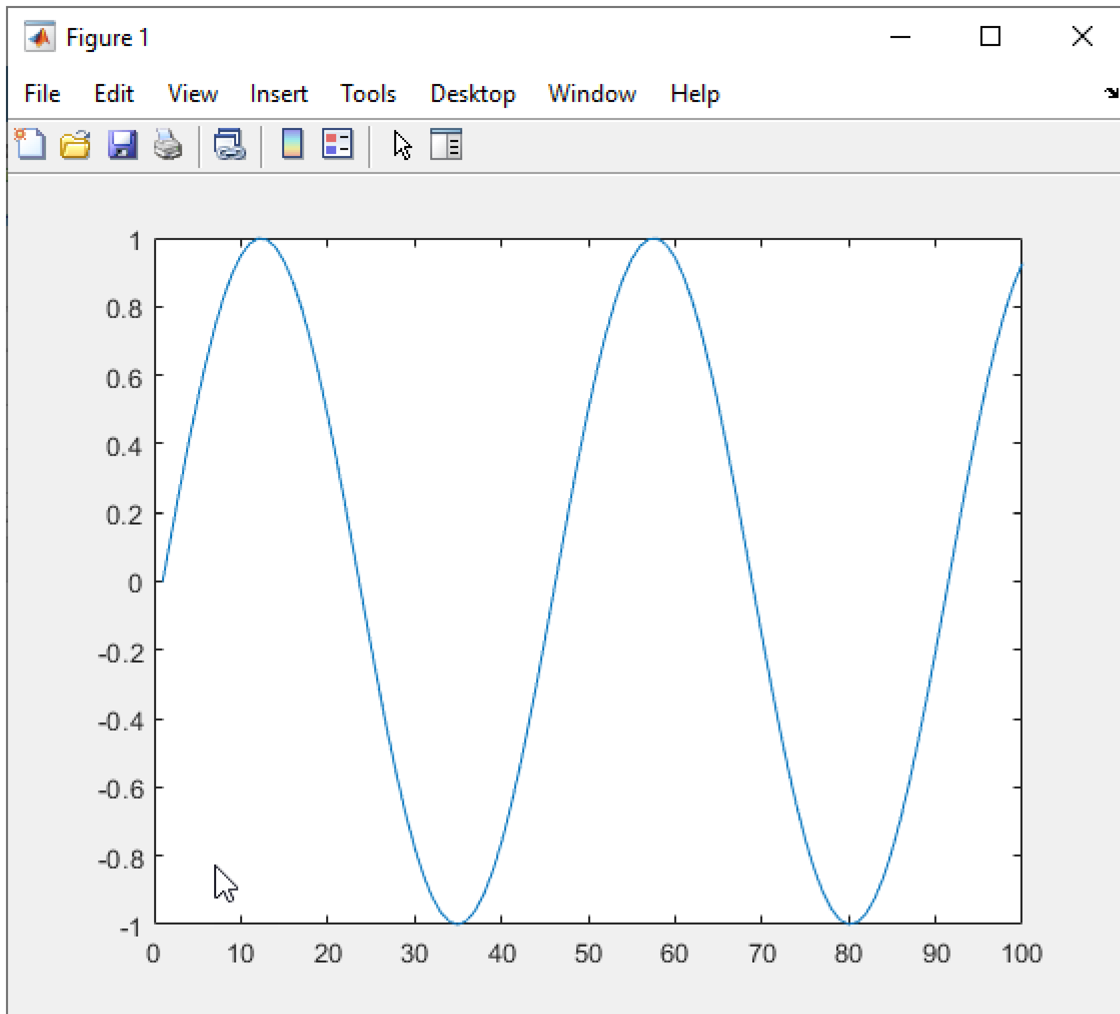
Communication Log

Action	Data	Size	Data Type	Time
WriteLine	Send Sine-Wave Data	1 x 1	string	22-Jun-2021 13:44:54
WriteLine	Send Arbitrary Waveform	1 x 1	string	22-Jun-2021 13:45:09
Read	0 0.13844 0.27422 0.40471 0.52742 0.63996 0.74018 0.82614 0.89619 0.94898 0.9835 0.99907 0.99541 0.97257 0.931 0.8715 0.79522 0.70362 0.59847 0.4818 0.35...	1 x 100	single	22-Jun-2021 13:49:38
Read	0 0.15048 0.2995 0.44562 0.58744 0.72359 0.85276 0.97372 1.0853 1.1866 1.2764 1.3542 1.4191 1.4706 1.5082 1.5318 1.5412 1.5363 1.5174 1.4848 1.4389 1.3802 ...	1 x 200	single	22-Jun-2021 13:50:46

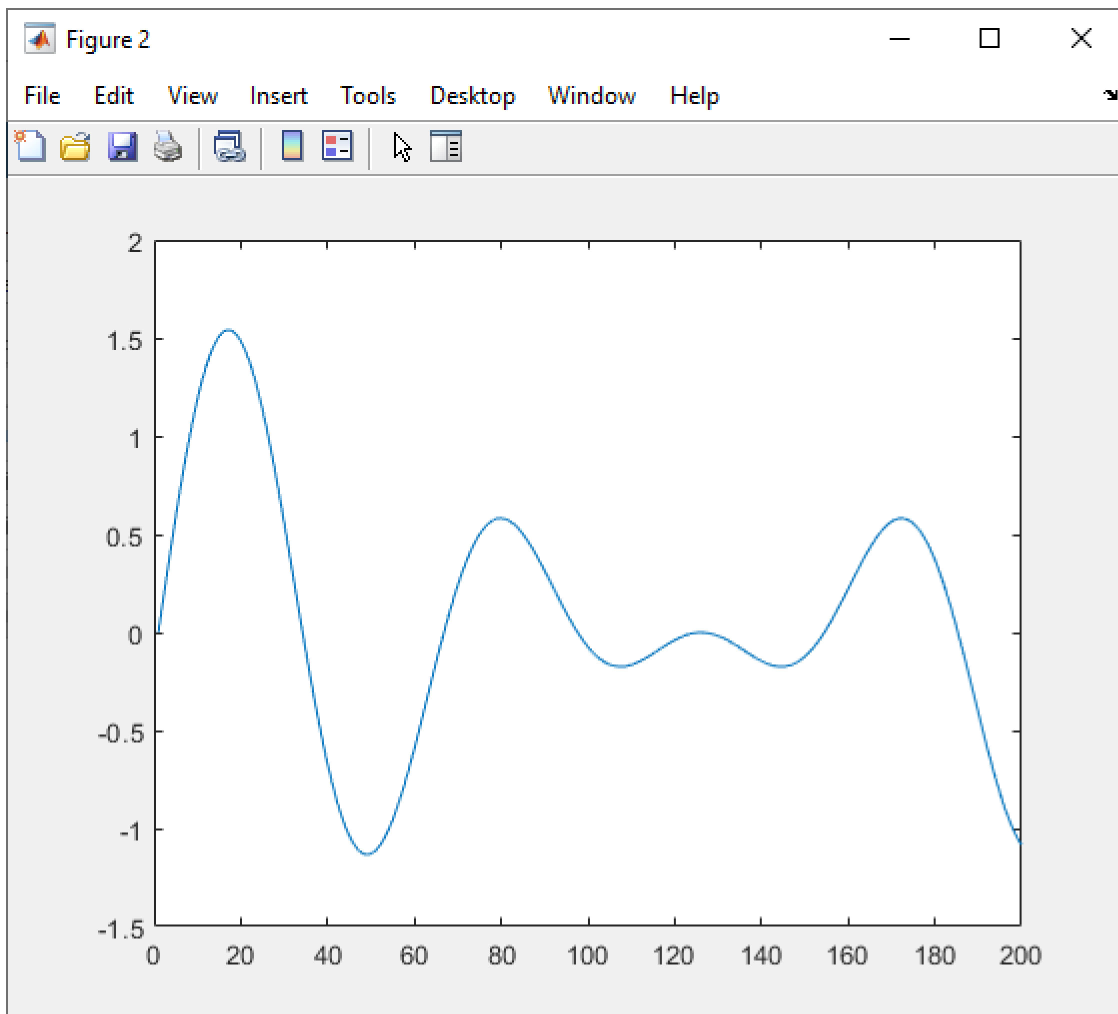
Click **Plot** in the **Analyze** section.



A new figure window with a plot of the data opens. You can modify the plot and figure from the command window.



Select the other response data and click **Plot** again. Another figure window with a plot of the data opens.



The **MATLAB Code Log** pane shows the code for these operations (except for plot creation). You can export this code as a MATLAB Live Script file by following the steps in “Export Data from Communication Log and Generate MATLAB Script” on page 24-17.

```
% Create a serialport object called serialportObj that connects to the port "COM4" with a
% default baud rate of 9600.
serialportObj = serialport("COM4",9600);

% Set the value of the "BaudRate" property of the serialport object serialportObj to
% 38400.
serialportObj.BaudRate = 38400;

% Write the data "Send Sine-Wave Data" as a string using the serialport object
% serialportObj. The write terminator "LF" is automatically appended to the data before writing.
writeline(serialportObj,"Send Sine-Wave Data");

% "command" is a workspace variable.
% Write the data command as a string using the serialport object serialportObj. The write
% terminator "LF" is automatically appended to the data before writing.
writeline(serialportObj,command);

% Read 100 values of single data using the serialport object serialportObj.
data1 = read(serialportObj,100,"single");

% Read 200 values of single data using the serialport object serialportObj.
data2 = read(serialportObj,200,"single");

```

Plot Data from Communication Log

In this example, plot a row of data from the **Communication Log** in a new figure window. You can plot any numeric data that you have written to or read from the serial port.

The **Communication Log** captures all the data that you have written to or read from the connected serial port.

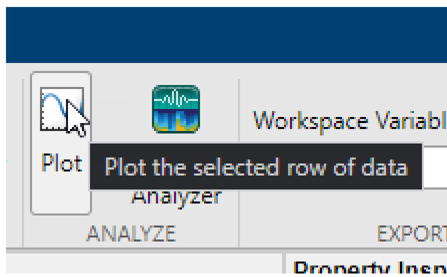
Select a row of data.

WRITE READ COMMUNICATION LOG ANALYZE

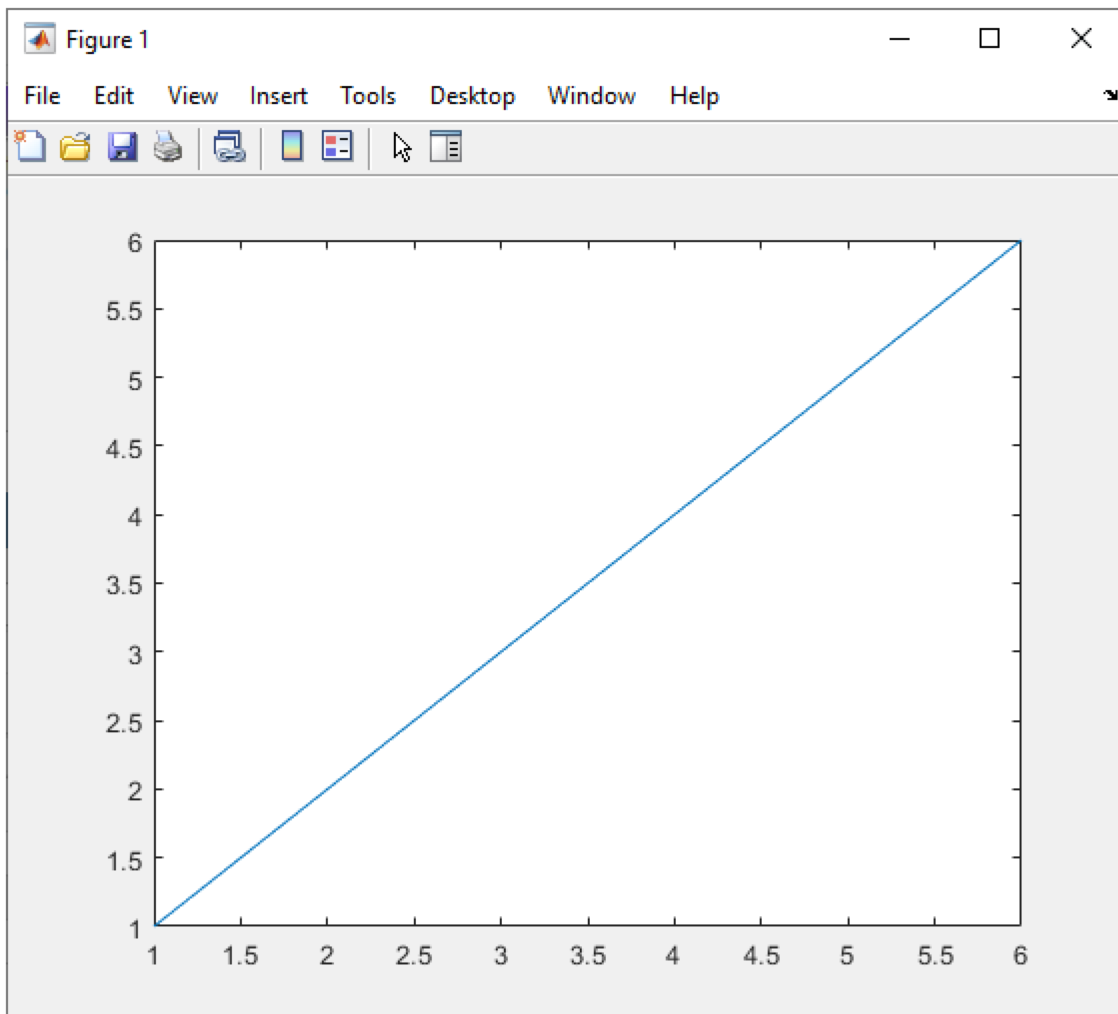
Communication Log

Action	Data	Size	Data Type	Time
Write	1 2 3 4 5 6 7 8 9 10	1 x 10	uint8	21-Jun-2021 10:48:16
Read	1 2 3 4 5 6	1 x 6	uint8	21-Jun-2021 10:49:58
Read	7 8 9 10	1 x 4	uint8	21-Jun-2021 10:54:38

Click **Plot** in the **Analyze** section.



A new figure window with a plot of the data opens. You can modify the plot and figure from the command window.



Export Data from Communication Log and Generate MATLAB Script

In this example, use the different options for exporting data and app interactions.

The **Communication Log** captures all the data that you have written to or read from the connected serial port.

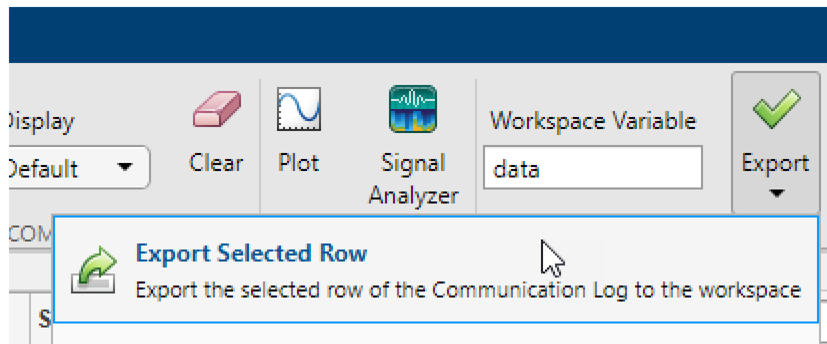
Select a row of data.

WRITE | READ | COMMUNICATION LOG | ANALYZE | EXPORT

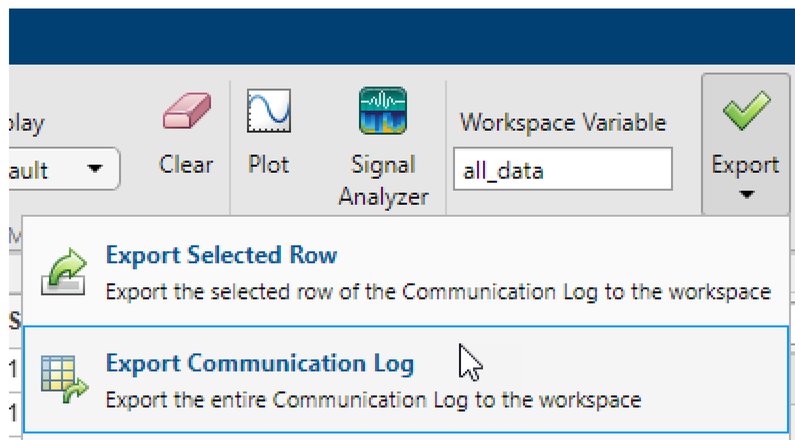
Communication Log

Action	Data	Size	Data Type	Time
WriteLine	Send Sine-Wave Data	1 x 1	string	22-Jun-2021 13:44:54
WriteLine	Send Arbitrary Waveform	1 x 1	string	22-Jun-2021 13:45:09
Read	0 0.13844 0.27422 0.40471 0.52742 0.63996 0.74018 0.82614 0.89619 0.94898 0.9835 0.99907 0.99541 0.97257 0.931 0.8715 0.79522 0.70362 0.59847 0.4818 0.35...	1 x 100	single	22-Jun-2021 13:49:38
Read	0 0.15048 0.2995 0.44562 0.58744 0.72359 0.85276 0.97372 1.0853 1.1866 1.2764 1.3542 1.4191 1.4706 1.5082 1.5318 1.5412 1.5363 1.5174 1.4848 1.4389 1.3802 ...	1 x 200	single	22-Jun-2021 13:50:46

Export this row of data to the workspace as the variable specified in **Workspace Variable**. The app provides a default variable name, but you can edit it. Change the variable name, click **Export**, and select the **Export Selected Row** option.



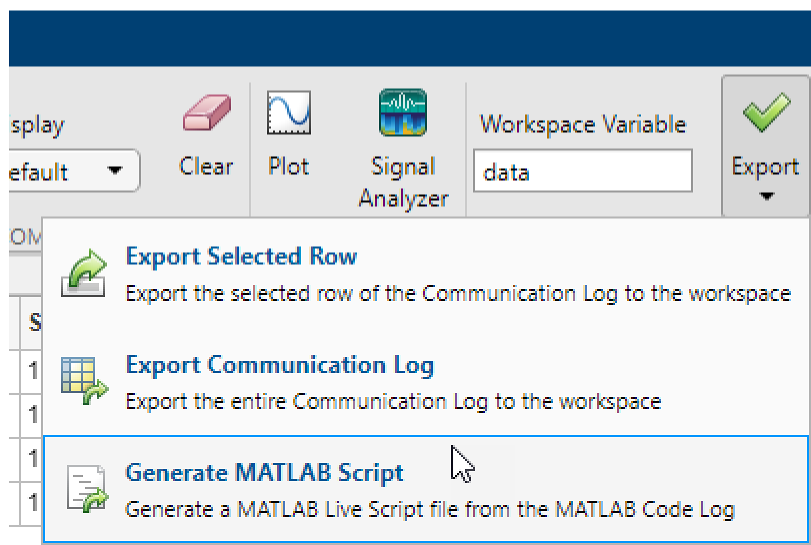
You can also export the entirety of the **Communication Log** to the workspace as a `timetable`. Change the variable name, click **Export**, and select the **Export Communication Log** option.



Besides exporting data, you can also export the code from the **MATLAB Code Log** pane. This pane contains all `serialport` object creation, write, read, and property configuration operations that you do in the app.

```
% Create a serialport object called serialportObj that connects to the port "COM4" with a
% default baud rate of 9600.
serialportObj = serialport("COM4",9600);

% Set the value of the "BaudRate" property of the serialport object serialportObj to
% 38400.
serialportObj.BaudRate = 38400;

% Write the data "Send Sine-Wave Data" as a string using the serialport object
% serialportObj. The write terminator "LF" is automatically appended to the data before writing.
writeline(serialportObj,"Send Sine-Wave Data");

% "command" is a workspace variable.
% Write the data command as a string using the serialport object serialportObj. The write
% terminator "LF" is automatically appended to the data before writing.
writeline(serialportObj,command);

% Read 100 values of single data using the serialport object serialportObj.
data1 = read(serialportObj,100,"single");

% Read 200 values of single data using the serialport object serialportObj.
data2 = read(serialportObj,200,"single");

```

Generate a MATLAB Live Script file and open it in the Live Editor by clicking **Export** and selecting the **Generate MATLAB Script** option.



After the Live Script file opens, you can modify the code to fit your needs and save the file.

Parameters

Write Section

`Data Format` — Select type of write operation
`Binary` (default) | `ASCII-Terminated String`

You can write `Binary` or `ASCII-Terminated String` data to the serial port.

A `Binary` write is equivalent to the `write` function and an `ASCII-Terminated String` write is equivalent to the `writeline` function.

`Data Type` — Select MATLAB data type to write
`uint8` | `int8` | `uint16` | `int16` | `uint32` | `int32` | `uint64` | `int64` | `single` | `double` | `char` | `string`

Specify the data type of the data to write to the serial port. This parameter determines the number of bytes to write for each value and the interpretation of those bytes as a MATLAB data type.

Dependencies

If you set the **Data Format** to `ASCII-Terminated String`, the only possible value for this parameter is `string`.

If you set the **Data Format** to `Binary`, the default value of this parameter is `uint8`.

This parameter can be set to `uint64` or `int64` only if you select the **Workspace Variable** option instead of **Data to Write**.

`Data to Write` — Specify numeric or ASCII data to write
numeric | character vector | string scalar

Specify the data to write to serial port. The data is written as the type specified by **Data Type**, regardless of the format in this parameter.

Select either this parameter or **Workspace Variable** to write data.

`Workspace Variable` — Select workspace variable to write
workspace variable

Select an existing workspace variable to write to the serial port. The data is written as the type specified by **Data Type**, regardless of the data type of the variable in the workspace.

If **Data Format** is `Binary`, you can select the following types of workspace variables:

- Row (1-by-N) or column (N-by-1) vector of numeric values
- 1-by-N character vector
- 1-by-1 string scalar

If **Data Format** is `ASCII-Terminated String`, you can select the following types of workspace variables:

- 1-by-N character vector
- 1-by-1 string scalar

Select either this parameter or **Data to Write** to write data.

`Write` — Write data using specified settings
button

Click this button to write the data specified in **Data to Write** or **Workspace Variable** to the serial port as the specified **Data Type**. If **Data Format** is `ASCII-Terminated String`, the write terminator specified by the **Terminator** property is automatically appended to the data being written.

This button is equivalent to performing the `write` or `writeline` functions.

Read Section

`Data Format` — Select type of read operation
`Binary` (default) | `ASCII-Terminated String`

Read `Binary` or `ASCII-Terminated String` data from the serial port. A `Binary` read is equivalent to the `read` function and an `ASCII-Terminated String` read is equivalent to the `readline` function.

`Data Type` — Select MATLAB data type to read
`uint8` | `int8` | `uint16` | `int16` | `uint32` | `int32` | `uint64` | `int64` | `single` | `double` | `char` | `string`

Specify the data type of the data to read from the serial port. This parameter determines the number of bytes to read for each value and the interpretation of those bytes as a MATLAB data type.

Dependencies

If you set the **Data Format** to `ASCII-Terminated String`, the only possible value for this parameter is `string`.

If you set the **Data Format** to `Binary`, the default value of this parameter is `uint8`.

`Num Values to Read` — Specify number of values of selected Data Type to read
numeric

Specify the number of values to read as a positive integer. This parameter must be less than or equal to **Values Available**. If you leave this parameter empty, the app reads all available values from the serial port using the specified **Data Type**.

Dependencies

To enable this parameter, set **Data Format** to `Binary`.

`Values Available` — Maximum possible number of values of selected Data Type that can be read
numeric

This property is read-only.

This is the number of values available to read in the format specified by **Data Type**.

Dependencies

To enable this parameter, set **Data Format** to `Binary`.

Read — Read data using specified settings
button

Click this button to read data from the serial port. If **Data Format** is `Binary`, read the number of values specified by **Num Values to Read** in the form specified by **Data Type**. If **Data Format** is `ASCII-Terminated String`, read data until the first occurrence of the read terminator specified by the **Terminator** property.

This button is equivalent to the `read` or `readline` functions.

Communication Log Section

`Display` — Select format to view data in Communication Log
`Default` (default) | `Binary` | `ASCII` | `Hexadecimal`

View the data in the **Data** column of the **Communication Log** as `Binary`, `ASCII`, or `Hexadecimal`, as applicable based on the data type. This parameter does not change the original value or data type of the data. For more information about these formats, see "Data Type Conversion".

`Clear` — Clear Communication Log
button

Click this button to clear all the contents of the **Communication Log**.

Analyze Section

`Plot` — Plot selected row of data
button

Click this button to open a new figure window that plots the data currently selected in the **Communication Log**. You can select only one row of data, and the selected data must be numeric.

Unlike **Write** and **Read**, this operation is not captured in the **MATLAB Code Log** pane.

`Signal Analyzer` — View selected row of data in Signal Analyzer app
button

Click this button to launch the **Signal Analyzer** app and send it the data currently selected in the **Communication Log**. You can select only one row of data, and the selected data must be a numeric vector.

You must have Signal Processing Toolbox™ installed to use the **Signal Analyzer** app.

Export Section

`Workspace Variable` — Specify name of workspace variable to export data to
valid variable name

Edit the name of the workspace variable that you want to export data to. The `Export Selected Row` and `Export Communication Log` options in **Export** save your data in the workspace as the variable specified by this parameter.

You must specify a valid MATLAB variable name that does not already exist in the workspace. If you specify an invalid name, it is automatically changed to a valid variable name.

Export — Export Communication Log data or MATLAB code

`Export Selected Row` | `Export Communication Log` | `Generate MATLAB Script`

Click this button to select one of the following options for exporting data from this app:

- `Export Selected Row` — Save the data currently selected in the **Communication Log** to the workspace as the variable specified by **Workspace Variable**.
- `Export Communication Log` — Save all of the **Communication Log** data to the workspace as a `timetable` with the variable name specified by **Workspace Variable**.
- `Generate MATLAB Script` — Generate a MATLAB Live Script file populated with the content in **MATLAB Code Log** and open it in the Live Editor.

Property Inspector

Port — Connected serial port

string scalar

This property is read-only.

Name of the connected serial port, returned as a character vector.

BaudRate — Communication speed

`9600` (default) | `1200` | `2400` | `4800` | `14400` | `19200` | `38400` | `57600` | `115200` | `230400` | `460800` | `500000` | `576000` | `921600` | `1000000` | numeric

Rate at which bits are transmitted for the serial interface, in bits per second. You can select one of the available options or specify your own value.

DataBits — Number of bits to represent one character of data

`8` (default) | `5` | `6` | `7`

Number of data bits to transmit over the serial interface.

StopBits — Pattern of bits that indicates end of character

`1` (default) | `1.5` | `2`

Number of bits used to indicate the end of a byte.

Parity — Parity bit type

`none` (default) | `even` | `odd`

Parity bit type added to data transmitted by serial port. You can use this parameter to add a parity bit (also referred to as a check bit) to your data. Adding a parity bit to a string of binary code is a method of detecting errors in data transmission by ensuring that the total number of 1-bits is even or odd.

The value of the parity bit is determined by the number of 1s in a given set of bits and is set as follows.

Parity Bit Type	Parity Bit Value	
	If number of 1s is even	If number of 1s is odd
none	No parity bit set	No parity bit set
even	0	1
odd	1	0

`FlowControl` — Mode for managing data transmission rate
`none` (default) | `hardware` | `software`

Process of managing the rate of data transmission on your serial port. Select `none` to have no flow control, `hardware` to let your hardware determine the flow control, or `software` to let your software determine the flow control.

`NumBytesAvailable` — Number of bytes available to read
numeric

This property is read-only.

Number of bytes available to read, returned as a numeric value.

`Terminator` — Terminator characters for data
`LF` (default) | `CR` | `CR/LF` | 0 to 255

Terminator characters for reading and writing ASCII-terminated data, specified as `LF`, `CR`, `CR/LF`, or a number from 0 to 255. The read terminator is followed by the write terminator and the two are separated by a comma. Click the vertical ellipsis icon to specify read and write terminator character values separately.

`ByteOrder` — Sequential order of bytes
`little-endian` (default) | `big-endian`

Sequential order in which bytes are arranged into larger numerical values. If the byte order is `little-endian`, then the serial port stores the first byte in the first memory address. If the byte order is `big-endian`, then the serial port stores the last byte in the first memory address.

Configure the byte order to match the appropriate value for your serial port.

`Timeout` — Allowed time to complete operations
10 (default) | numeric

Allowed time in seconds to complete read operations, specified as a numeric value.

See Also

Apps
TCP/IP Explorer

Functions
`serialport`

Introduced in R2021b

TCP/IP Explorer

Connect to and communicate with TCP/IP server

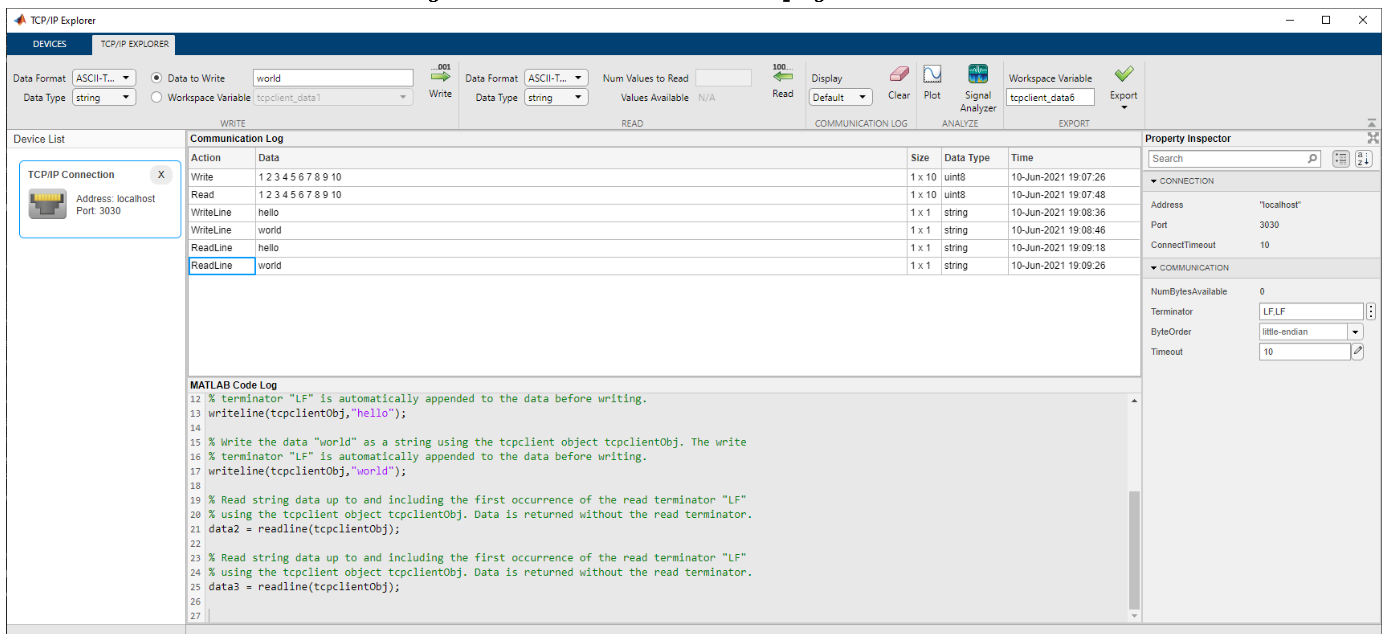
Description

The **TCP/IP Explorer** app creates a TCP/IP client connection to an existing TCP/IP server. After you connect to a server, you can communicate with it, plot and analyze data, export data to the workspace, and generate MATLAB code.

Using this app, you can:

- Configure TCP/IP communication properties.
- Send binary or string data from the TCP/IP client to the connected server.
- Read binary or string data sent to the TCP/IP client from the connected server.
- Plot data in a figure window.
- Analyze data by viewing it in the **Signal Analyzer** app.
- Export data to the workspace.
- Generate a MATLAB Live Script file that uses the `tcpclient` interface.

You can use this app only as a client and not as a server. For information on creating a TCP/IP server, see "Communicate Using TCP/IP Server Sockets" on page 7-34.



Open the TCP/IP Explorer App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app icon.

- MATLAB command prompt: Enter `tcpipExplorer`.

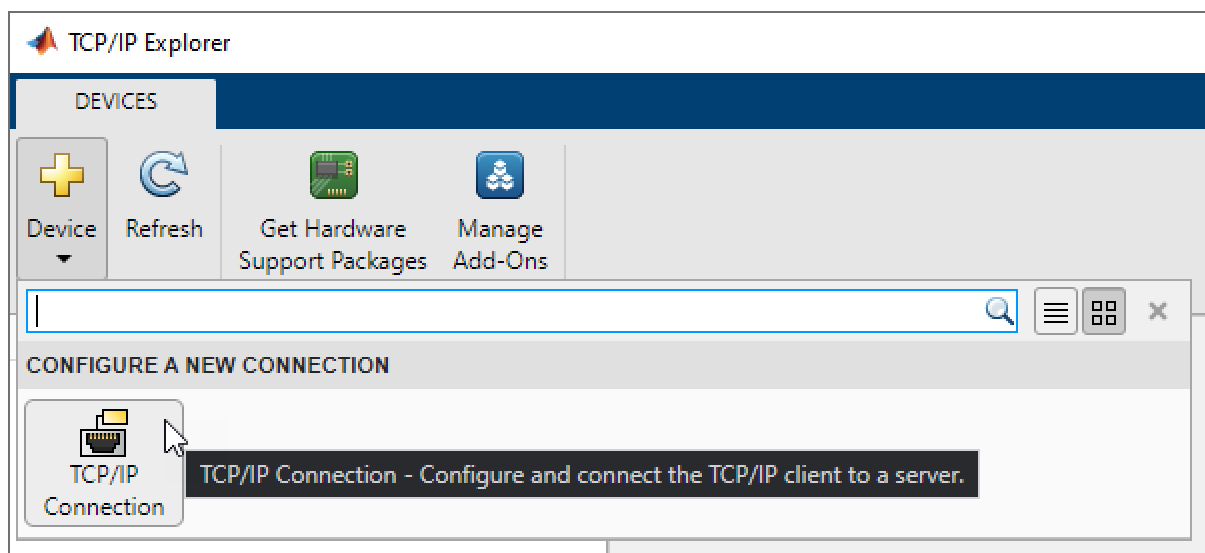
Examples

Connect TCP/IP Client to TCP/IP Server

In this example, connect to a TCP/IP server.

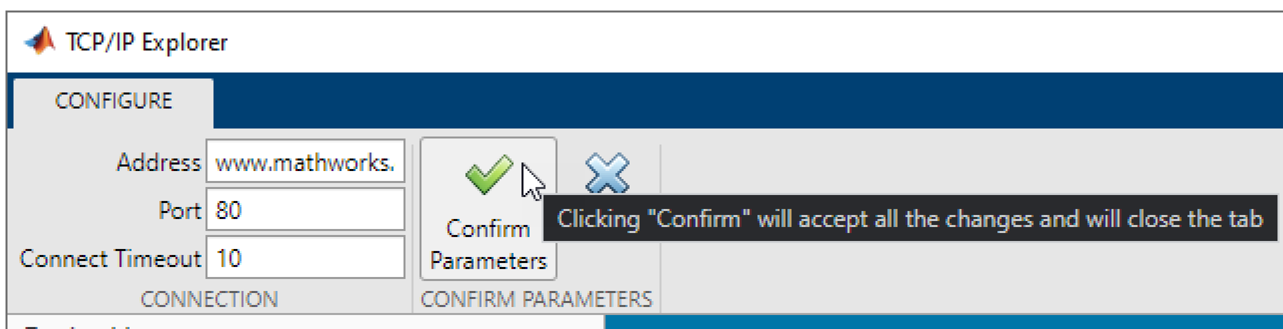
Open the **TCP/IP Explorer** app from either the **Apps** tab in the MATLAB toolstrip or the MATLAB command prompt.

On the **Devices** tab in the app, click **Device > TCP/IP Connection**.



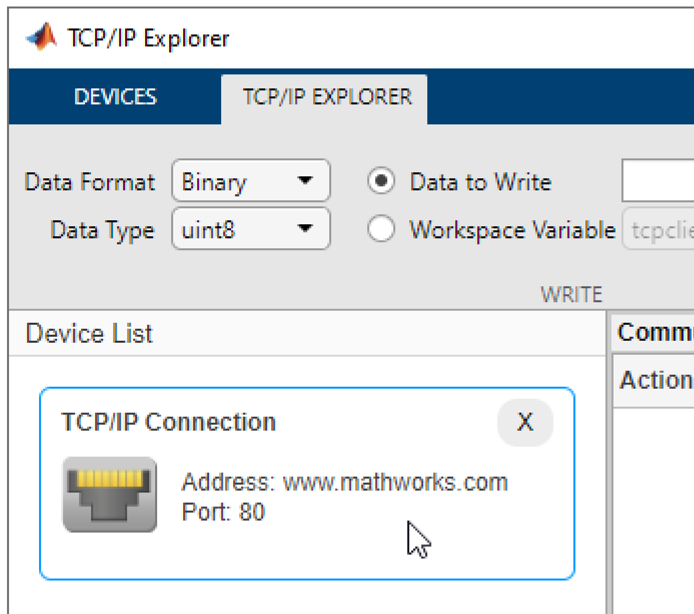
Specify **Address** as the server host name `www.mathworks.com` and **Port** as the server port `80`. You can leave the **Connect Timeout** as the default value of `10`. For more information about these parameters, see “Configure Connection in TCP/IP Explorer” on page 7-11.

Click **Confirm Parameters** to create a TCP/IP client connected to the specified TCP/IP server.

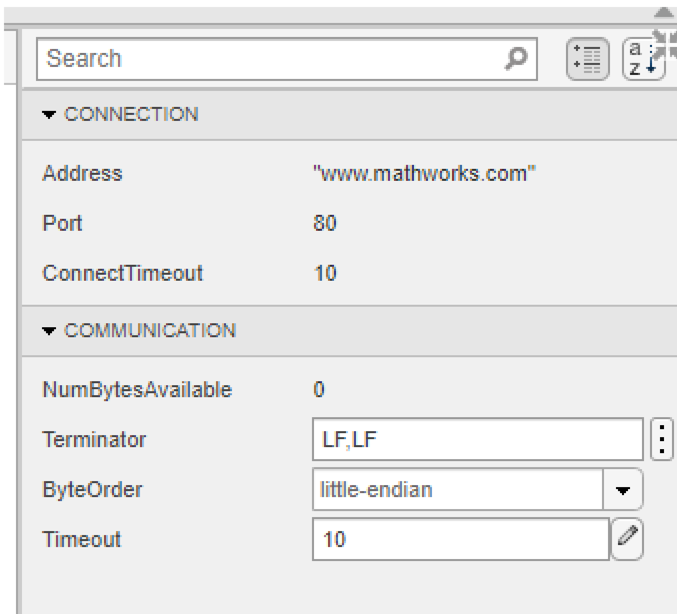


Alternatively, you can specify **Address** as the server IP address. In this example, the IP address for the host name is `144.212.130.17`.

The **TCP/IP Explorer** tab opens in the app and your client **TCP/IP Connection** appears in the **Device List**.



View **Connection** properties and configure **Communication** properties from the **Property Inspector**. You can specify the **Terminator**, **ByteOrder**, and **Timeout** properties of the server before writing and reading data.

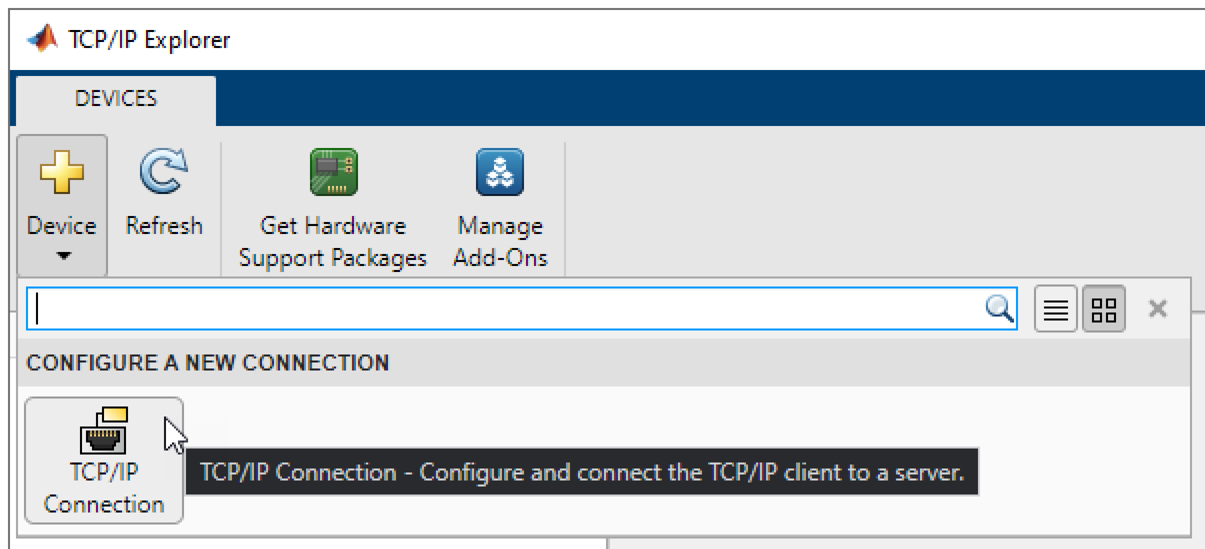


Write ASCII-Terminated Data to TCP/IP Client and Read Response

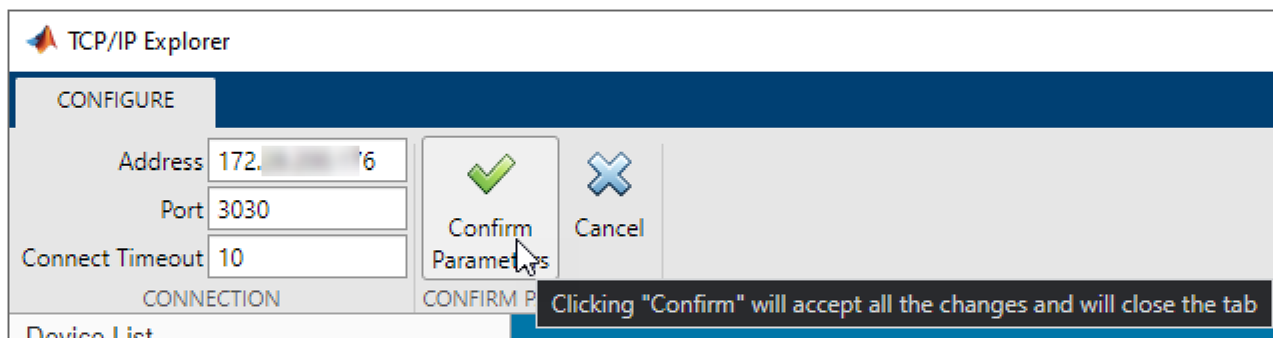
In this example, write ASCII-terminated data to a TCP/IP client connected to a server and read data back from it. The TCP/IP server in this example has already been programmed with custom commands and responses.

Open the **TCP/IP Explorer** app from either the **Apps** tab in the MATLAB toolstrip or the MATLAB command prompt.

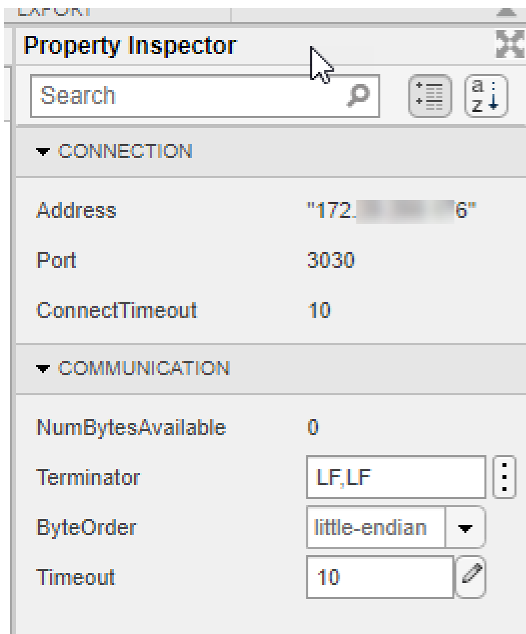
On the **Devices** tab in the app, click **Device > TCP/IP Connection**.



Specify **Address** as the server address and **Port** as the server port to connect to the server. The values specified in this example are specific to this server and do not work on other machines. You can leave the **Connect Timeout** as the default value of `10`. Click **Confirm Parameters** to create a TCP/IP client connected to the specified server.

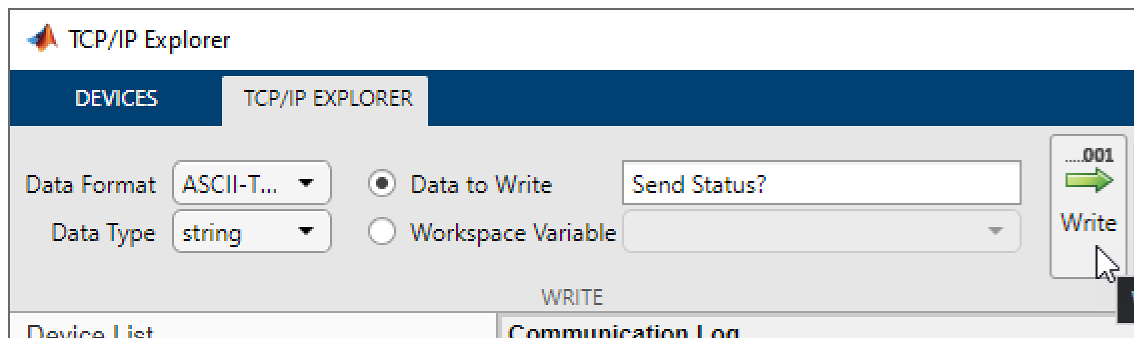


Before writing or reading data, you can modify **Communication** properties from the **Property Inspector**. Ensure that these properties match the appropriate values for the server. For this example, the values shown already match the server configuration.

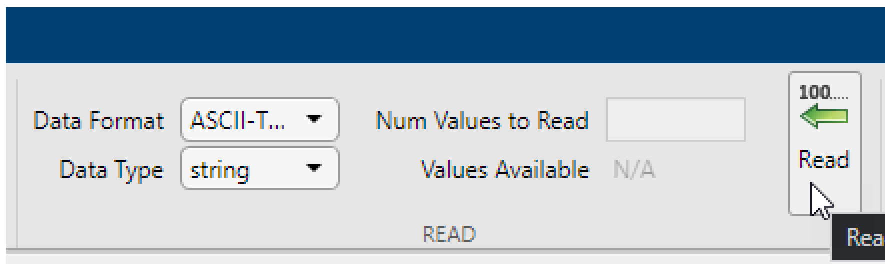


Some TCP/IP clients can accept string queries to send to the server and respond to them. In this example, the connected server has been programmed to receive and respond to customized string commands. The commands in this example do not work for other clients.

Send the `Send Status?` command from the client to the server. In the **Write** section, set the **Data Format** to `ASCII-Terminated String`. The **Data Type** changes to `string` since that is the only possible option. Specify the **Data to Write** as `Send Status?`. Click **Write** to write the data from the client to the server. For ASCII-terminated string write operations, the write terminator specified by the **Terminator** property is automatically appended to the data being written.



You can view the response to this command by reading from the client. In the **Read** section, set the **Data Format** to `ASCII-Terminated String`, which changes the **Data Type** to `string`. Read the data sent to the client from the server by clicking **Read**. Data is read until the first occurrence of a terminator.



View both the write and read operations in the **Communication Log** pane. The read operation shows the message `Server Running on "172.XX.XXX.XXX" and port 3030.`. You can select a row to export it as a variable to the workspace by following the steps in “Export Data from Communication Log and Generate MATLAB Script” on page 24-39.

Communication Log

Action	Data	Size	Data Type	Time
WriteLine	Send Status?	1 x 1	string	02-Jul-2021 12:49:13
ReadLine	Server Running on "172.[illegible]6" and port 3030.	1 x 1	string	02-Jul-2021 12:49:17

The **MATLAB Code Log** pane shows the code for these operations. You can export this code as a MATLAB Live Script file by following the steps in “Export Data from Communication Log and Generate MATLAB Script” on page 24-39.

MATLAB Code Log

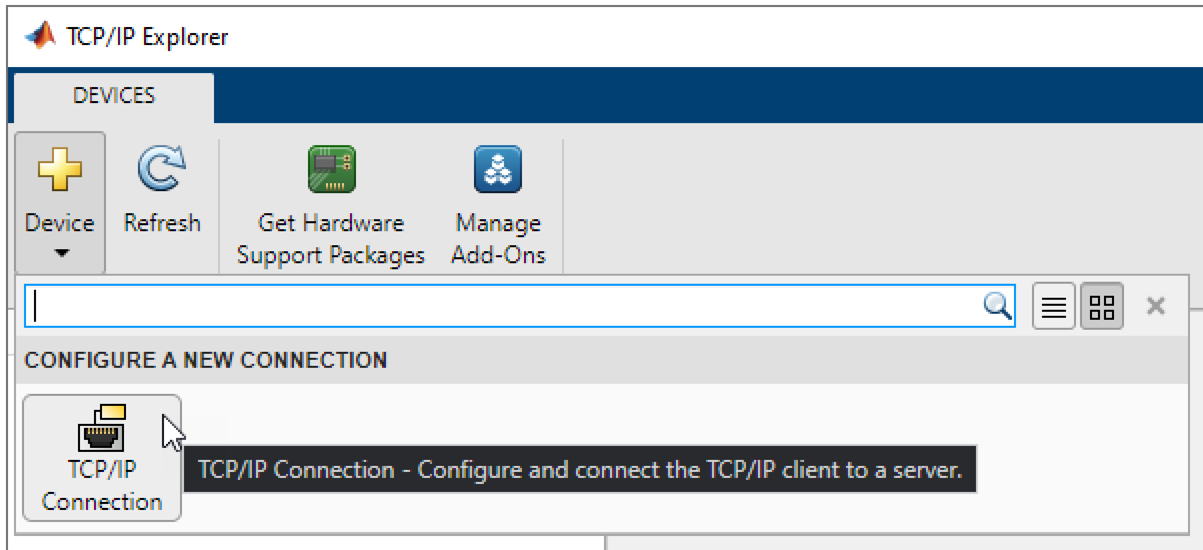
```
1  % Create a tcpclient object tcpclientObj that connects to IP address or host name
2  % "172.[illegible]6" and port 3030 with a connection timeout period of 10 seconds.
3  tcpclientObj = tcpclient("172.[illegible]6",3030,"ConnectTimeout",10);
4
5  % Write the data "Send Status?" as a string using the tcpclient object tcpclientObj. The
6  % write terminator "LF" is automatically appended to the data before writing.
7  writeline(tcpclientObj,"Send Status?");
8
9  % Read string data up to and including the first occurrence of the read terminator "LF"
10 % using the tcpclient object tcpclientObj. Data is returned without the read terminator.
11 data1 = readline(tcpclientObj);
12
13
```

Read Binary Data from TCP/IP Client and Plot It

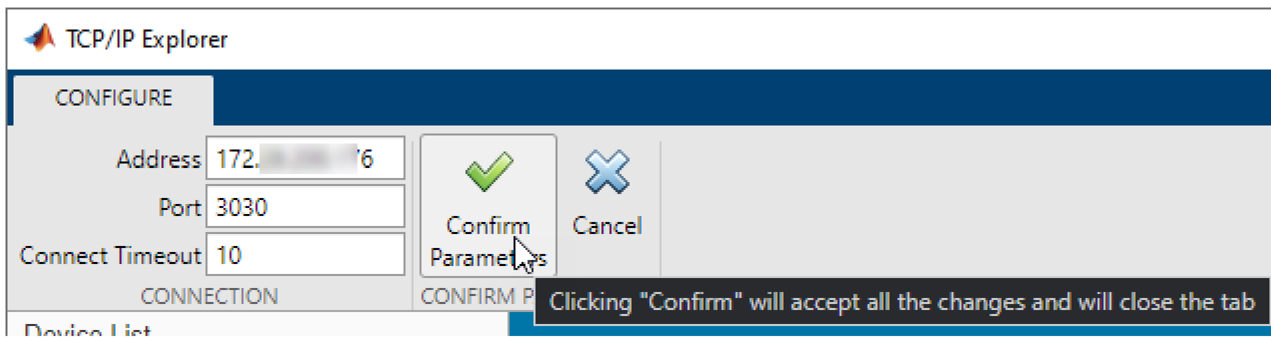
In this example, write ASCII-terminated data to a TCP/IP client connected to a server and read data back from it. The TCP/IP server is this example has already been programmed with custom commands and responses.

Open the **TCP/IP Explorer** app from either the **Apps** tab in the MATLAB toolstrip or the MATLAB command prompt.

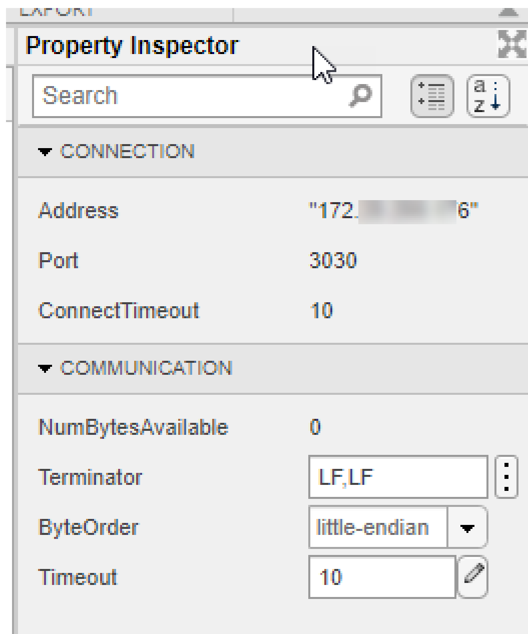
On the **Devices** tab in the app, click **Device > TCP/IP Connection**.



Specify **Address** as the server address and **Port** as the server port to connect to the server. The values specified in this example are specific to this server and do not work on other machines. You can leave the **Connect Timeout** as the default value of `10`. Click **Confirm Parameters** to create a TCP/IP client connected to the specified server.

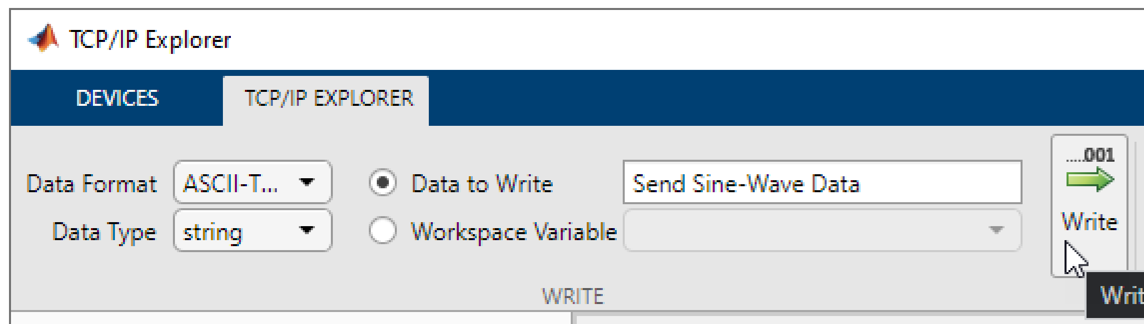


Before writing or reading data, you can modify **Communication** properties from the **Property Inspector**. Ensure that these properties match the appropriate values for the server. For this example, the values shown already match the server configuration.



Some TCP/IP clients can accept string queries to send to the server and respond to them. In this example, the connected server has been programmed to receive and respond to customized string commands. The commands in this example do not work for other clients.

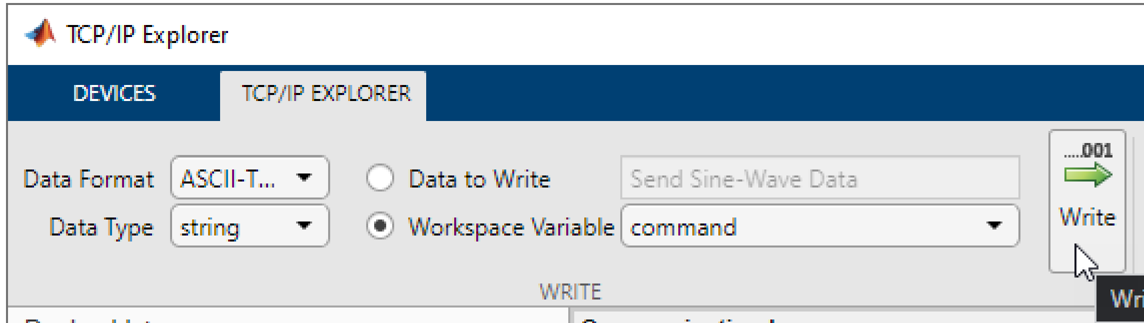
Send the `Send Sine-Wave Data` command from the client to the server. In the **Write** section, set the **Data Format** to `ASCII-Terminated String`. The **Data Type** changes to `string` since that is the only possible option. Specify the **Data to Write** as `Send Sine-Wave Data`. Click **Write** to write the data from the client to the server. For ASCII-terminated string write operations, the write terminator specified by the **Terminator** property is automatically appended to the data being written.



Send another command to the client. In the MATLAB command prompt, create a workspace variable for this command.

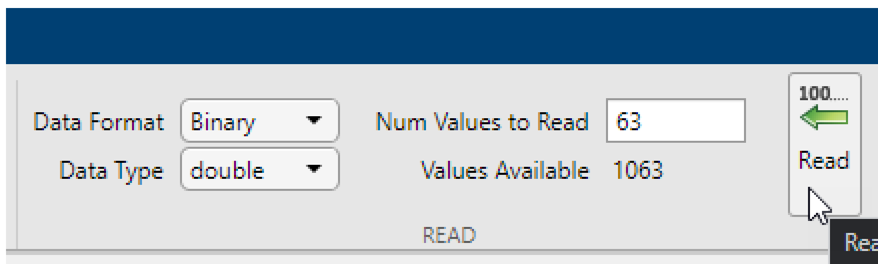
```
command = "Send Arbitrary Waveform";
```

Select **Workspace Variable** and select the `command` option. Click **Write**.

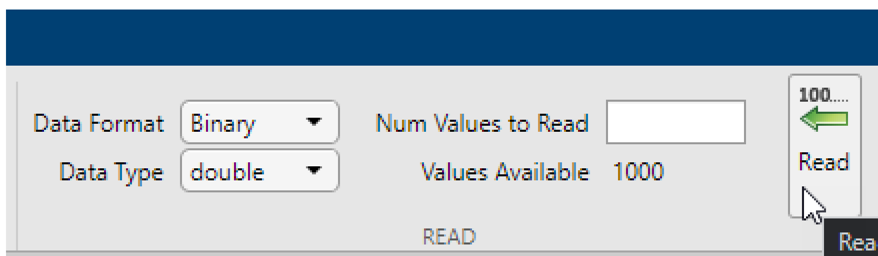


Before reading data from the client, you must specify the correct data format and type. In this example, the responses to the string commands are stored as binary data with double precision. In the **Read** section, set the **Data Format** to `Binary`, and the **Data Type** to `double`. The **Values Available** parameter is `1063`. The first 63 values are the response to the `Send Sine-Wave Data` command and the remaining 1000 values are the response to the `Send Arbitrary Waveform` command.

Specify the **Num Values to Read** as `63`. Read the first 63 values of the data from the client by clicking **Read**.



If you do not specify a value for the **Num Values to Read** parameter, you can read all the available values. Read the remaining 1000 values by clearing the **Num Values to Read** parameter and clicking **Read**.

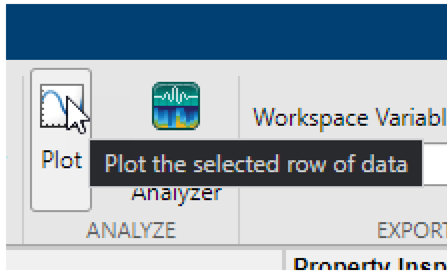


View both the write and read operations in the **Communication Log** pane. You can select a row to plot it, view it in the **Signal Analyzer** app, or export it as a variable to the workspace. Select the data from the first read operation.

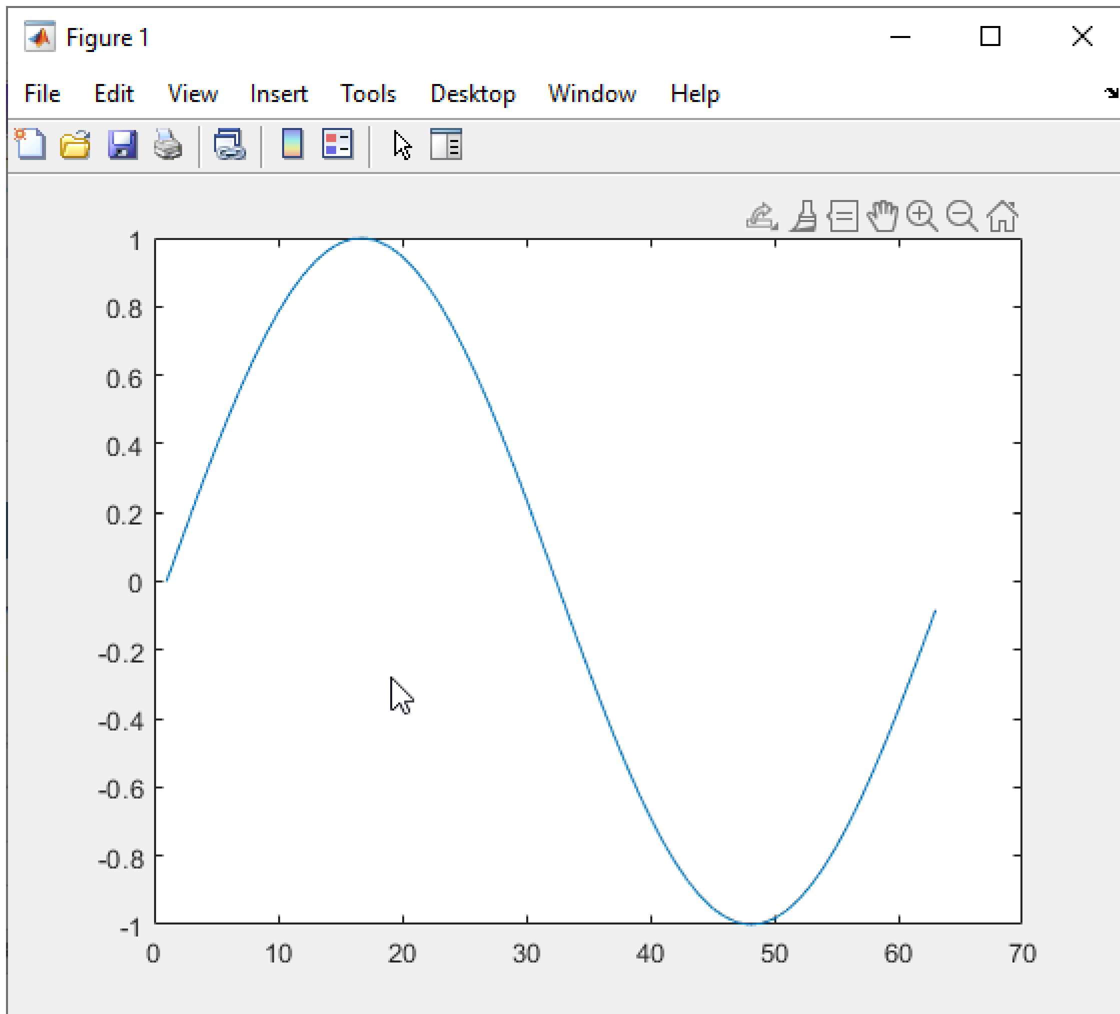
Communication Log

Action	Data	Size	Data Type	Time
WriteLine	Send Sine-Wave Data	1 x 1	string	06-Jul-2021 11:33:55
WriteLine	Send Arbitrary Waveform	1 x 1	string	06-Jul-2021 11:33:58
Read	0 0.099833 0.19867 0.29552 0.38942 0.47943 0.56464 0.64422 0.71736 0.78333 0.84147 0.89121 0.93204 0.96356 0.98545...	1 x 63	double	06-Jul-2021 11:52:37
Read	0.016377 0.20529 0.27167 0.25902 0.3453 0.56366 0.55468 0.54288 0.60988 0.68036 0.85506 0.82706 0.87873 0.89617 0....	1 x 1000	double	06-Jul-2021 11:53:19

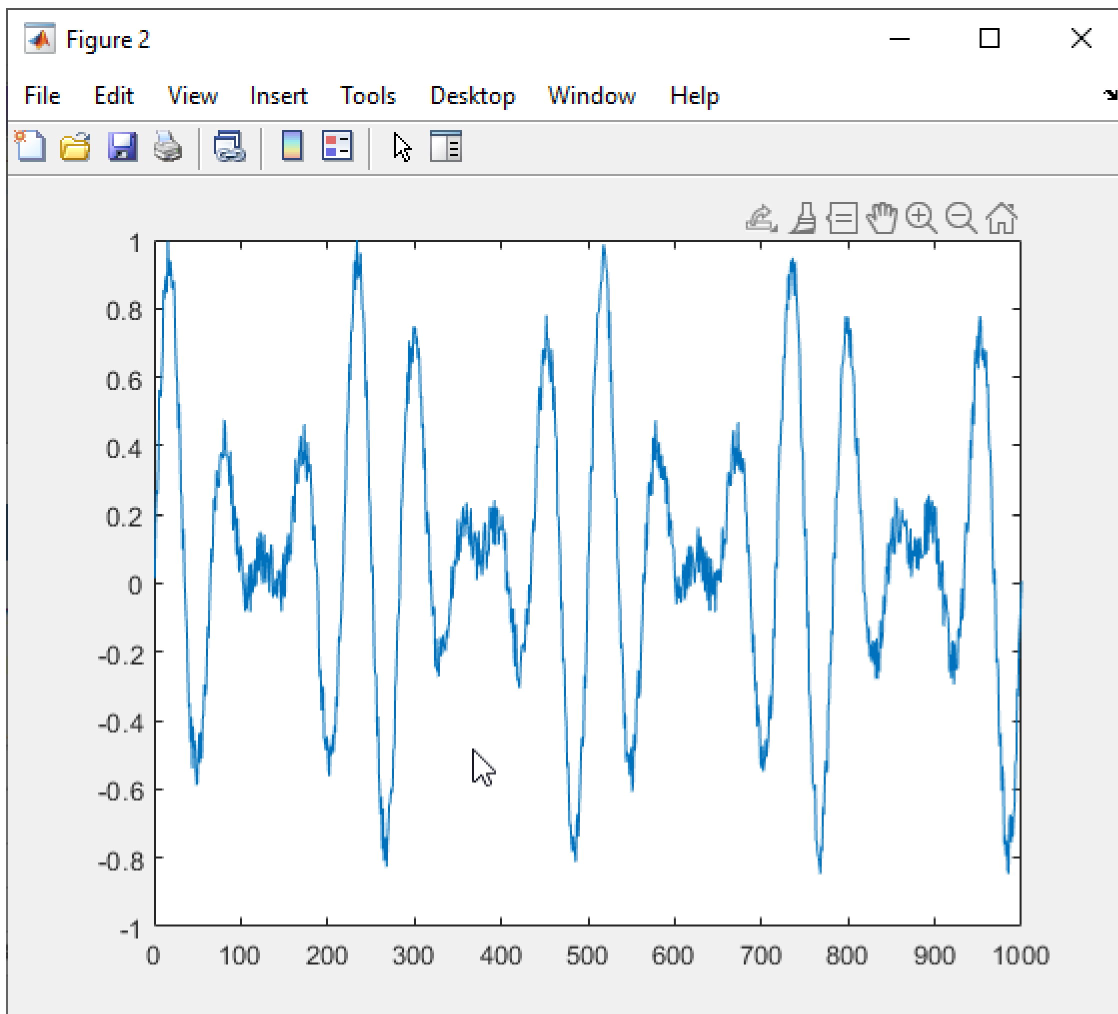
Click **Plot** in the **Analyze** section.



A new figure window with a plot of the data opens. You can modify the plot and figure from the command window.



Select the other response data and click **Plot** again. Another figure window with a plot of the data opens.



The **MATLAB Code Log** pane shows the code for these operations (except for plot creation). You can export this code as a MATLAB Live Script file by following the steps in “Export Data from Communication Log and Generate MATLAB Script” on page 24-39.

MATLAB Code Log

```
1  % Create a tcpclient object tcpclientObj that connects to IP address or host name
2  % "172.[illegible]6" and port 3030 with a connection timeout period of 10 seconds.
3  tcpclientObj = tcpclient("172[illegible]6",3030,"ConnectTimeout",10);
4
5  % Write the data "Send Sine-Wave Data" as a string using the tcpclient object
6  % tcpclientObj. The write terminator "LF" is automatically appended to the data before writing.
7  writeline(tcpclientObj,"Send Sine-Wave Data");
8
9  % "command" is a workspace variable.
10 % Write the data command as a string using the tcpclient object tcpclientObj. The write
11 % terminator "LF" is automatically appended to the data before writing.
12 writeline(tcpclientObj,command);
13
14 % Read 63 values of double data using the tcpclient object tcpclientObj.
15 data1 = read(tcpclientObj,63,"double");
16
17 % Read 1000 values of double data using the tcpclient object tcpclientObj.
18 data2 = read(tcpclientObj,1000,"double");
19
20
```

Plot Data from Communication Log

In this example, plot a row of data from the **Communication Log** in a new figure window. You can plot any numeric data that you have written to or read from the TCP/IP server.

The **Communication Log** captures all the data that you have written to or read from the connected TCP/IP server.

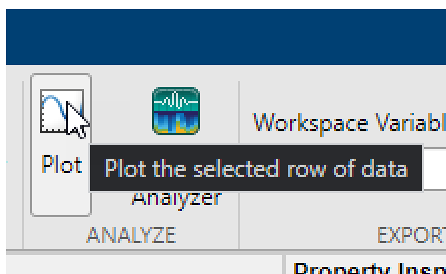
Select a row of data.

WRITE | READ | COMMUNICATION LOG | ANALYZE

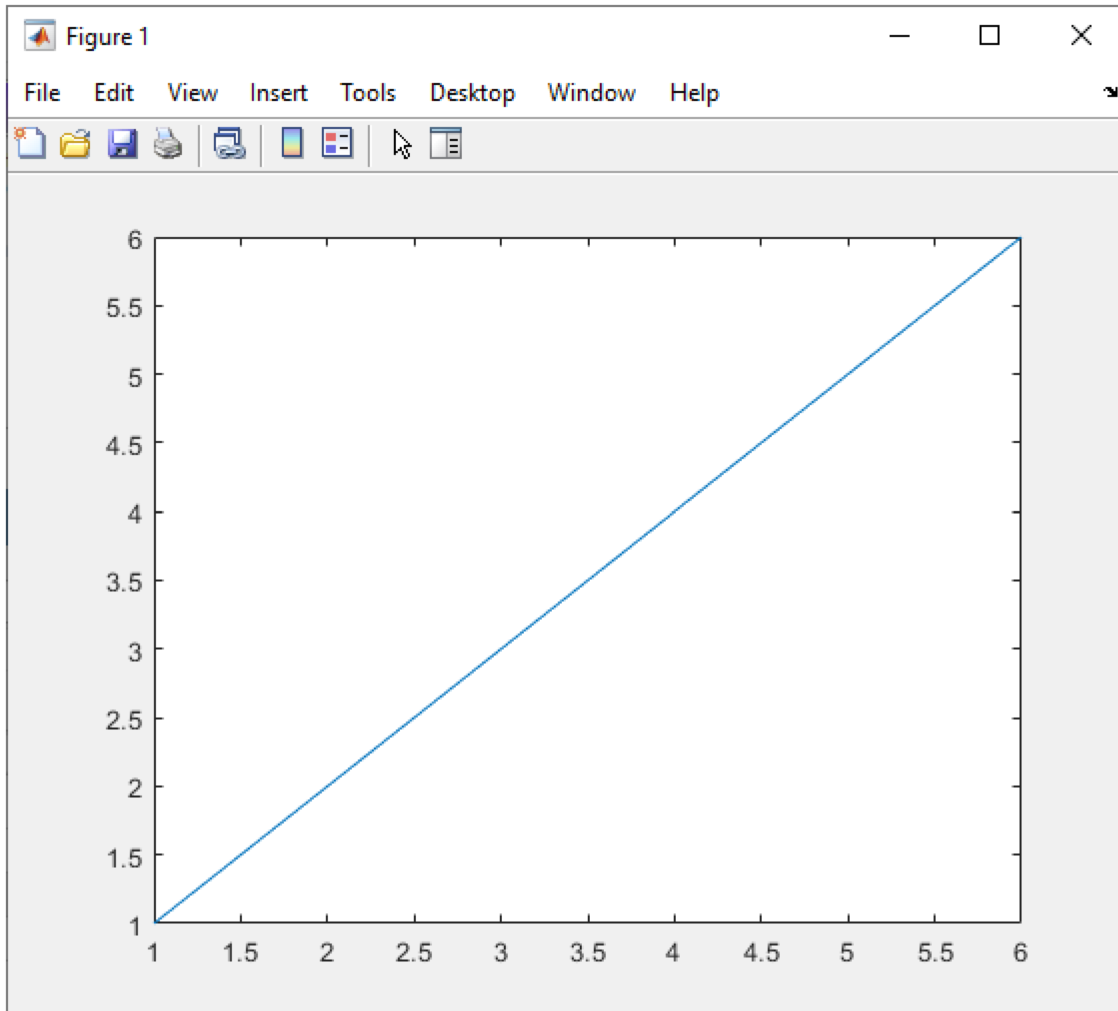
Communication Log

Action	Data	Size	Data Type	Time
Write	1 2 3 4 5 6 7 8 9 10	1 x 10	uint8	21-Jun-2021 10:48:16
Read	1 2 3 4 5 6	1 x 6	uint8	21-Jun-2021 10:49:58
Read	7 8 9 10	1 x 4	uint8	21-Jun-2021 10:54:38

Click **Plot** in the **Analyze** section.



A new figure window with a plot of the data opens. You can modify the plot and figure from the command window.



Export Data from Communication Log and Generate MATLAB Script

In this example, use the different options for exporting data and app interactions.

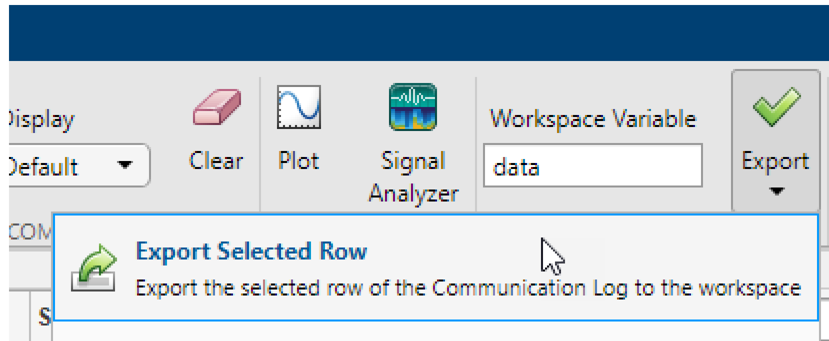
The **Communication Log** captures all the data that you have written to or read from the connected TCP/IP server.

Select a row of data.

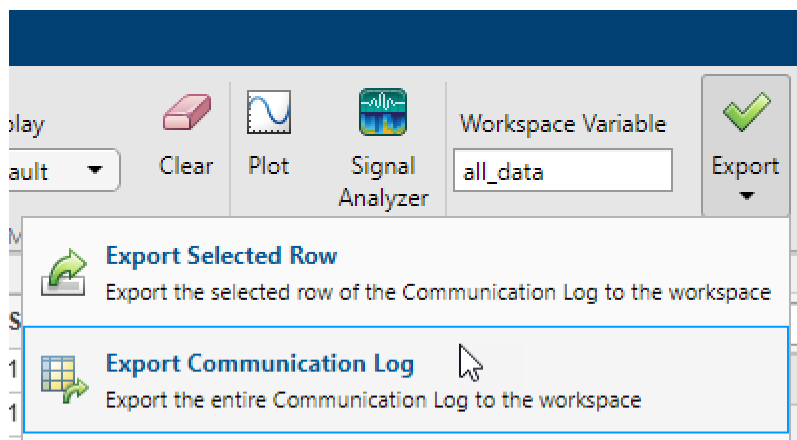
Communication Log

Action	Data	Size	Data Type	Time
WriteLine	Send Status?	1 x 1	string	02-Jul-2021 12:49:13
ReadLine	Server Running on "172. 6" and port 3030.	1 x 1	string	02-Jul-2021 12:49:17

Export this row of data to the workspace as the variable specified in **Workspace Variable**. The app provides a default variable name, but you can edit it. The data is saved in the workspace as its **Data Type**. Change the variable name, click **Export**, and select the **Export Selected Row** option.



You can also export the entirety of the **Communication Log** to the workspace as a `timetable`. Change the variable name, click **Export**, and select the **Export Communication Log** option.

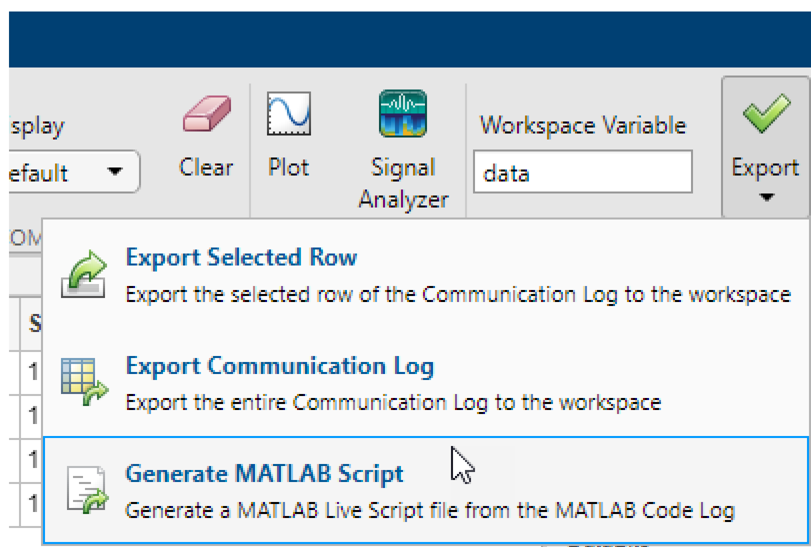


Besides exporting data, you can also export the code from the **MATLAB Code Log** pane. This pane contains all `tcpclient` object creation, write, read, and property configuration operations that you do in the app.

MATLAB Code Log

```
1  % Create a tcpclient object tcpclientObj that connects to IP address or host name
2  % "172.[illegible]6" and port 3030 with a connection timeout period of 10 seconds.
3  tcpclientObj = tcpclient("172.[illegible]6",3030,"ConnectTimeout",10);
4
5  % Write the data "Send Status?" as a string using the tcpclient object tcpclientObj. The
6  % write terminator "LF" is automatically appended to the data before writing.
7  writeline(tcpclientObj,"Send Status?");
8
9  % Read string data up to and including the first occurrence of the read terminator "LF"
10 % using the tcpclient object tcpclientObj. Data is returned without the read terminator.
11 data1 = readline(tcpclientObj);
12
13
```

Generate a MATLAB Live Script file and open it in the Live Editor by clicking **Export** and selecting the **Generate MATLAB Script** option.



After the Live Script file opens, you can modify the code to fit your needs and save the file.

Parameters

Write Section

`Data Format` — Select type of write operation

`Binary` (default) | `ASCII-Terminated String`

You can write `Binary` or `ASCII-Terminated String` data from the TCP/IP client to the connected server.

A `Binary` write is equivalent to the `write` function and an `ASCII-Terminated String` write is equivalent to the `writeline` function.

`Data Type` — Select MATLAB data type to write

`uint8` | `int8` | `uint16` | `int16` | `uint32` | `int32` | `uint64` | `int64` | `single` | `double` | `char` | `string`

Specify the data type of the data to write from the TCP/IP client to the connected server. This parameter determines the number of bytes to write for each value and the interpretation of those bytes as a MATLAB data type.

Dependencies

If you set the **Data Format** to `ASCII-Terminated String`, the only possible value for this parameter is `string`.

If you set the **Data Format** to `Binary`, the default value of this parameter is `uint8`.

This parameter can be set to `uint64` or `int64` only if you select the **Workspace Variable** option instead of **Data to Write**.

`Data to Write` — Specify numeric or ASCII data to write

numeric | character vector | string scalar

Specify the data to write from the TCP/IP client to the connected server. The data is written as the type specified by **Data Type**, regardless of the format in this parameter.

Select either this parameter or **Workspace Variable** to write data.

`Workspace Variable` — Select workspace variable to write

workspace variable

Select an existing workspace variable to write from the TCP/IP client to the connected server. The data is written as the type specified by **Data Type**, regardless of the data type of the variable in the workspace.

If **Data Format** is `Binary`, you can select the following types of workspace variables:

- Row (1-by-N) or column (N-by-1) vector of numeric values
- 1-by-N character vector
- 1-by-1 string scalar

If **Data Format** is `ASCII-Terminated String`, you can select the following types of workspace variables:

- 1-by-N character vector
- 1-by-1 string scalar

Select either this parameter or **Data to Write** to write data.

`Write` — Write data using specified settings

button

Click this button to write the data specified in **Data to Write** or **Workspace Variable** from the TCP/IP client to the connected server as the specified **Data Type**. If **Data Format** is `ASCII-`

`Terminated String`, the write terminator specified by the **Terminator** property is automatically appended to the data being written.

This button is equivalent to performing the `write` or `writeline` functions.

Read Section

`Data Format` — Select type of read operation

`Binary` (default) | `ASCII-Terminated String`

Read `Binary` or `ASCII-Terminated String` data received by the TCP/IP client from the connected server. A `Binary` read is equivalent to the `read` function and an `ASCII-Terminated String` read is equivalent to the `readline` function.

`Data Type` — Select MATLAB data type to read

`uint8` | `int8` | `uint16` | `int16` | `uint32` | `int32` | `uint64` | `int64` | `single` | `double` | `char` | `string`

Specify the data type of the data received by the TCP/IP client from the connected server. This parameter determines the number of bytes to read for each value and the interpretation of those bytes as a MATLAB data type.

Dependencies

If you set the **Data Format** to `ASCII-Terminated String`, the only possible value for this parameter is `string`.

If you set the **Data Format** to `Binary`, the default value of this parameter is `uint8`.

`Num Values to Read` — Specify number of values of selected Data Type to read

numeric

Specify the number of values to read as a positive integer. This parameter must be less than or equal to **Values Available**. If you leave this parameter empty, the client reads all available values using the specified **Data Type**.

Dependencies

To enable this parameter, set **Data Format** to `Binary`.

`Values Available` — Maximum possible number of values of selected Data Type that can be read

numeric

This property is read-only.

This is the number of values available to read in the format specified by **Data Type**.

Dependencies

To enable this parameter, set **Data Format** to `Binary`.

Read — Read data using specified settings

button

Click this button to read data received by the TCP/IP client from the connected server. If **Data Format** is `Binary`, read the number of values specified by **Num Values to Read** in the form

specified by **Data Type**. If **Data Format** is `ASCII-Terminated String`, read data until the first occurrence of the read terminator specified by the **Terminator** property.

This button is equivalent to the `read` or `readline` functions.

Communication Log Section

`Display` — Select format to view data in Communication Log

`Default` (default) | `Binary` | `ASCII` | `Hexadecimal`

View the data in the **Data** column of the **Communication Log** as `Binary`, `ASCII`, or `Hexadecimal`, as applicable based on the data type. This parameter does not change the original value or data type of the data. For more information about these formats, see “Data Type Conversion”.

`Clear` — Clear Communication Log

button

Click this button to clear all the contents of the **Communication Log**.

Analyze Section

`Plot` — Plot selected row of data

button

Click this button to open a new figure window that plots the data currently selected in the **Communication Log**. You can select only one row of data, and the selected data must be numeric.

Unlike **Write** and **Read**, this operation is not captured in the **MATLAB Code Log** pane.

`Signal Analyzer` — View selected row of data in Signal Analyzer app

button

Click this button to launch the **Signal Analyzer** app and send it the data currently selected in the **Communication Log**. You can select only one row of data, and the selected data must be a numeric vector.

You must have Signal Processing Toolbox installed to use the **Signal Analyzer** app.

Export Section

`Workspace Variable` — Specify name of workspace variable to export data to

valid variable name

Edit the name of the workspace variable that you want to export data to. The `Export Selected Row` and `Export Communication Log` options in **Export** save your data in the workspace as the variable specified by this parameter.

You must specify a valid MATLAB variable name that does not already exist in the workspace. If you specify an invalid name, it is automatically changed to a valid variable name.

`Export` — Export Communication Log data or MATLAB code

`Export Selected Row` | `Export Communication Log` | `Generate MATLAB Script`

Click this button to select one of the following options for exporting data from this app:

- `Export Selected Row` — Save the data currently selected in the **Communication Log** to the workspace as the variable specified by **Workspace Variable**. The data is saved as its **Data Type**.

- `Export Communication Log` — Save all of the **Communication Log** data to the workspace as a `timetable` with the variable name specified by **Workspace Variable**.
- `Generate MATLAB Script` — Generate a MATLAB Live Script file populated with the content in **MATLAB Code Log** and open it in the Live Editor.

Property Inspector

`Address` — Server name or IP address
character vector

This property is read-only.

Server name or IP address, returned as a character vector. This property is set during TCP/IP Connection configuration.

`Port` — Server port
numeric

This property is read-only.

Server port, returned as a number between 1 and 65535, inclusive. This property is set during TCP/IP Connection configuration.

`ConnectTimeout` — Allowed time to connect to server
`10` (default) | numeric

This property is read-only.

Allowed time in seconds to connect to the server, specified as a numeric value. This property specifies the maximum time to wait for a connection request to the specified server to succeed or fail. This property is set during TCP/IP Connection configuration.

`NumBytesAvailable` — Number of bytes available to read
numeric

This property is read-only.

Number of bytes available to read, returned as a numeric value.

`Terminator` — Terminator characters for data
`LF` (default) | `CR` | `CR/LF` | 0 to 255

Terminator characters for reading and writing ASCII-terminated data, specified as `LF`, `CR`, `CR/LF`, or a number from 0 to 255. The read terminator is followed by the write terminator and the two are separated by a comma. Click the vertical ellipsis icon to specify read and write terminator character values separately.

`ByteOrder` — Sequential order of bytes
`little-endian` (default) | `big-endian`

Sequential order in which bytes are arranged into larger numerical values. If the byte order is `little-endian`, then the remote server stores the first byte in the first memory address. If the byte order is `big-endian`, then the remote server stores the last byte in the first memory address.

Configure the byte order to match the appropriate value for your server.

`Timeout` — Allowed time to complete operations
10 (default) | numeric

Allowed time in seconds to complete read operations, specified as a numeric value.

See Also

Apps
Serial Explorer

Functions
`tcpclient`

Topics
“Configure Connection in TCP/IP Explorer” on page 7-11

Introduced in R2021b

add

Add entry to IVI configuration store object

Syntax

```
add(obj, 'type', 'name', ...)
add(obj, 'DriverSession', 'name', 'ModuleName', 'HardwareAssetName', 'P1',
V1)
add(obj, 'HardwareAsset', 'name', 'IOResourceDescriptor', 'P1', V1)
add(obj, 'LogicalName', 'name', 'SessionName', 'P1', V1)
add(obj, struct)
```

Arguments

obj	IVI configuration store object
'**DriverSession**' '**HardwareAsset**' '**LogicalName**'	Type of entry being added
'name'	Name of the DriverSession, HardwareAsset, or LogicalName being added
'IOResourceDescriptor'	Tells the driver exactly how to locate the device this asset represents
'ModuleName'	IVI instrument driver or software module
'HardwareAssetName'	Unique identifier for hardware asset
'SessionName'	Unique identifier for asset driver session
'*P1*'	First optional parameter for added entry. Other parameter-value pairs may follow.
V1	Value for first parameter
struct	Structure defining entry to be added; field names are the entry parameter names

Description

add(obj, '*type*', 'name', ...) adds a new entry of *type* to the IVI configuration store object, obj, with name, name. If an entry of type, *type*, with name, name, already exists an error will occur. Based on *type*, additional arguments are required. *type* can be HardwareAsset, DriverSession, or LogicalName.

add(obj, 'DriverSession', 'name', 'ModuleName', 'HardwareAssetName', '*P1*', V1) adds a new driver session entry to the IVI configuration store object, obj, with name, name, using the specified software module name, ModuleName and hardware asset name, HardwareAssetName. Optional parameter-value pairs may be included.

Valid parameters for DriverSession are listed below. The default value for on/off parameters is off.

Parameter	Value	Description
`Description`	Any character vector	Description of driver session
`VirtualNames`	structure	A struct array containing virtual name mappings
`Cache`	`on`/`off`	Enable caching if the driver supports it.
`DriverSetup`	Any character vector	This value is software module dependent
`InterchangeCheck`	`on`/`off`	Enable driver interchangeability checking, if supported
`QueryInstrStatus`	`on`/`off`	Enable instrument status querying by the driver
`RangeCheck`	`on`/`off`	Enable extended range checking by the driver, if supported
`RecordCoercions`	`on`/`off`	Enable recording of coercions by the driver, if supported
`Simulate`	`on`/`off`	Enable simulation by the driver

`add(obj, 'HardwareAsset', 'name', 'IOResourceDescriptor', 'P1', V1)` adds a new hardware asset entry to the IVI configuration store object, `obj`, with name, `name`, and resource descriptor, `IOResourceDescriptor`. Optional parameter-value pairs may be included.

Valid parameters for `HardwareAsset` are

Parameter	Value	Description
`Description`	Any character vector	Description of hardware asset

`add(obj, 'LogicalName', 'name', 'SessionName', 'P1', V1)` adds a new logical name entry to the IVI configuration store object, `obj`, with name, `name`, and driver session name, `SessionName`. Optional parameter-value pairs may be included.

Valid parameters for `LogicalName` are

Parameter	Value	Description
`Description`	Any character vector	Description of logical name

`add(obj, struct)`, where `struct` is a structure whose field names are the entry parameter names, adds an entry to the IVI configuration store object, `obj`, of the specified type with the values contained in the structure.

Additions made to the configuration store object, `obj`, can be saved to the configuration store data file with the `commit` function.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For

information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Examples

Construct IVI configuration store object, `c`.

```
c = iviconfigurationstore;
```

Add a hardware asset with name `gpib1`, and resource description `GPIB0::1::INSTR`.

```
add(c, 'HardwareAsset', 'gpib1', 'GPIB0::1::INSTR');
```

Add a driver session with name `S1`, that uses the TekScope software module and the hardware asset with name `gpib1`.

```
add(c, 'DriverSession', 'S1', 'TekScope', 'gpib1');
```

Add a logical name to configuration store object `c`, with name `MyScope`, driver session name `S1`, and description `A logical name`.

```
add(c, 'LogicalName', 'MyScope', 'S1', ...
'Description', 'A logical name');
```

Add a hardware asset with the name `gpib3`, and resource description `GPIB0::3::ISNTR`.

```
s.Type = 'HardwareAsset';
s.Name = 'gpib3';
s.IOResourceDescriptor = 'GPIB0::3::INSTR';
add(c, s);
```

Save the changes to the IVI configuration store data file.

```
commit(c);
```

See Also

`iviconfigurationstore` | `commit` | `remove` | `update`

Introduced before R2006a

binblockread

(To be removed) Read binblock data from instrument

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see "Compatibility Considerations".

Syntax

```
A = binblockread(obj)
A = binblockread(obj,'precision')
[A,count] = binblockread(...)
[A,count,msg] = binblockread(...)
```

Arguments

`obj`	An interface object.
`'precision'`	The number of bits read for each value, and the interpretation of the bits as character, integer, or floating-point values.
`A`	Binblock data returned from the instrument.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`A = binblockread(obj)` reads binary-block (binblock) data from the instrument connected to `obj` and returns the values to `A`. The binblock format is described in the `binblockwrite` reference pages.

`A = binblockread(obj,'precision')` reads binblock data translating the MATLAB values to the precision specified by *`precision`*. By default the `uchar` precision is used and numeric values are returned in double-precision arrays. Refer to the `fread` function for a list of supported precisions.

`[A,count] = binblockread(...)` returns the number of values read to `count`.

`[A,count,msg] = binblockread(...)` returns a warning message to `msg` if the read operation did not complete successfully.

Examples

Create the GPIB object `g` associated with a National Instruments GPIB controller with board index 0, and a Tektronix TDS 210 oscilloscope with primary address 2.

```
g = gpib('ni',0,2);
g.InputBufferSize = 3000;
```

Connect g to the instrument, and write string commands that configure the scope to transfer binary waveform data from memory location A.

```
fopen(g)
fprintf(g,'DATA:DESTINATION REFA');
fprintf(g,'DATA:ENCDG SRPbinary');
fprintf(g,'DATA:WIDTH 1');
fprintf(g,'DATA:START 1');
```

Write the CURVE? command, which prepares the scope to transfer data, and read the data using the binblock format.

```
fprintf(g,'CURVE?')
data = binblockread(g);
```

If the scope sends a terminating character after the binblock, binblockread does not read the terminating character. Read it by using fread. In this example, count is the number of bytes of the terminating character and can be 1 or 2.

```
if g.BytesAvailable == count
   fread(g,count,'uint8');
end
```

Tips

Before you can read data from the instrument, it must be connected to obj with the fopen function. A connected interface object has a Status property value of open. An error is returned if you attempt to perform a read operation while obj is not connected to the instrument.

binblockread blocks the MATLAB Command Window until one of the following occurs:

- The data is completely read.
- The time specified by the Timeout property passes.

If msg is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

Each time binblockread is issued, the ValuesReceived property value is increased by the number of values read.

Some instruments may send a terminating character after the binblock. binblockread will not read the terminating character. You can read the terminating character with the fread function. Additionally, if obj is a GPIB, VISA-GPIB, VISA-VXI, VISA-USB, or VISA-RSIB object, you can use the clrdevice function to remove the terminating character.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

`udp` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See "Transition Your Code to udpport Interface" on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed

Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed

Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See "Transition Your Code to VISA-GPIB Interface" on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`binblockwrite` | `fopen` | `fread` | `instrhelp`

Properties
BytesAvailable | InputBufferSize | Status | ValuesReceived

Introduced before R2006a

binblockwrite

(To be removed) Write binblock data to instrument

Note This serial, Bluetooth, tcpip, udp, visa, and gpib object function will be removed in a future release. Use serialport, bluetooth, tcpclient, tcpserver, udpport, and visadev object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
binblockwrite(obj,A)
binblockwrite(obj,A,'precision')
binblockwrite(obj,A,'header')
binblockwrite(obj,A,'precision','header')
binblockwrite(obj,A,'precision','header','headerformat')
```

Arguments

obj	An interface object.
A	The data to be written using the binblock format.
'*precision*'	The number of bits written for each value, and the interpretation of the bits as character, integer, or floating-point values.
'header'	The ASCII header text to be prefixed to the data.
'*headerformat*'	C language conversion specification format for the header text.

Description

binblockwrite(obj,A) writes the data specified by A to the instrument connected to obj as a binary-block (binblock). The binblock format is defined as #<N><D><A>, where

- N specifies the number of digits in D that follow.
- D specifies the number of data bytes in A that follow.
- A is the data written to the instrument.

For example, if A is given by [0 5 5 0 5 5 0], the binblock would be defined as [double('#') 1 7 0 5 5 0 5 5 0].

binblockwrite(obj,A,'*precision*') writes binblock data translating the MATLAB values to the precision specified by *precision*. By default the uchar precision is used. Refer to the fwrite function for a list of supported precisions.

binblockwrite(obj,A,'header') writes a binblock using the data, A, and the ASCII header, header, to the instrument connected to interface object, obj. The data written is constructed using the formula

```
<header>#<N><D><A>
```

`binblockwrite(obj,A,'precision','header')` writes binary data, `A`, translating the MATLAB values to the specified precision, *`precision`*. The ASCII header, `header`, is prefixed to the binblock.

`binblockwrite(obj,A,'precision','header','headerformat')` writes binary data, `A`, translating the MATLAB values to the specified precision, *`precision`*. The ASCII header, `header`, is prefixed to the binblock using the format specified by *`headerformat`*.

`headerformat` is a string containing C language conversion specifications. Conversion specifications are composed of the character `%` and the conversion characters `d`, `i`, `o`, `u`, `x`, `X`, `f`, `e`, `E`, `g`, `G`, `c`, and `s`. Type `instrhelp fprintf` for more information on valid values for *`headerformat`*. By default, *`headerformat`* is `%s`.

Examples

```
s = visa('ni', 'ASRL2::INSTR');
fopen(s);

% Write the command: [double('#14') 0 5 0 5] to the instrument.
binblockwrite(s, [0 5 0 5]);

% Write the command: [double('Curve #14') 0 5 0 5] to the
% instrument.
binblockwrite(s, [0 5 0 5], 'Curve ')
fclose(s);
```

Tips

Before you can write data to the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the instrument.

The `ValuesSent` property value is increased by the number of values written each time `binblockwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See “Transition Your Code to bluetooth Interface” for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See “Transition Your Code to tcpclient Interface” on page 7-36 or “Transition Your Code to tcpserver Interface” on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See “Transition Your Code to udpport Interface” on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

gpib object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`binblockread` | `fopen` | `fwrite` | `instrhelp`

Properties
OutputBufferSize | OutputEmptyFcn | Status | Timeout | TransferStatus | ValuesSent

Introduced before R2006a

Bluetooth

(To be removed) Create Bluetooth object

Note `Bluetooth` will be removed in a future release. Use `bluetooth` (case-sensitive) instead. For more information, see "Compatibility Considerations".

Syntax

```
B = Bluetooth(RemoteName,Channel)
B = Bluetooth(RemoteID,Channel)
B = Bluetooth( ___ ,Name,Value)
```

Description

`B = Bluetooth(RemoteName,Channel)` creates a Bluetooth object associated with the `RemoteName` and `Channel`. `RemoteName` is a friendly way to identify the `RemoteID`. If not specified, the default channel is `0`.

The Instrument Control Toolbox Bluetooth interface lets you connect to devices over the Bluetooth interface, and to transmit and receive ASCII and binary data. Instrument Control Toolbox supports the Bluetooth Serial Port Profile (SPP). You can identify any SPP Bluetooth device and establish a two-way connection with that device.

`B = Bluetooth(RemoteID,Channel)` creates a Bluetooth object directly from the `RemoteID` and `Channel`.

To connect with the Bluetooth device, use the `fopen` function. When the Bluetooth object is created, its `status` property is `closed`. When the object is connected to the remote device with the `fopen` function, the `status` property is set to `open`.

`B = Bluetooth( ___ ,Name,Value)` creates a Bluetooth object using the specified property values. If an invalid property name or property value is specified the object is not created.

Examples

Write and Read with a Bluetooth Device

This example shows how to identify and connect to a Bluetooth device, send a message, and read data.

Find available Bluetooth devices.

```
instrhwinfo('Bluetooth')
```

Create a Bluetooth object called `b` using channel `3` of a Lego Mindstorm robot with a `RemoteName` of `NXT`.

```
b = Bluetooth('NXT',3);
```

Connect to the remote device.

```
fopen(b)
```

Send a message to the remote device.

```
fwrite(b,uint8([2,0,1,155]));
```

Read data from the remote device.

```
name = fread(b,35);
```

Disconnect the device.

```
fclose(b);
```

Clean up by deleting and clearing the object.

```
delete(b)
clear b
```

Input Arguments

RemoteName — Name for Bluetooth device

char | string

"Friendly name" for the Bluetooth device, specified as a character vector or string. For example, in the case of an iPhone, it might be simply `'iPhone'` or a name like `'Zor'`. If it is empty, use the `RemoteID` to communicate with the device.

Example: `'NXT'`

Data Types: `char` | `string`

RemoteID — Internal ID of Bluetooth device

char | string

Internal ID of the Bluetooth device, equivalent to the Device ID, specified as a character vector or string. Every device has a device ID, which is usually a 12-digit character vector that starts with `'btspp://'`. You can use this or the `RemoteName` to communicate with the device.

Example: `'btspp://0016530FD65F'`

Data Types: `char` | `string`

Channel — Device channel

0 (default) | numeric value

Device channel, specified as a numeric value, if the device has channels. If no channel is specified, it defaults to `0`.

Example: 3

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

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

Example: `'Timeout',60`

Name — Name of interface object
char | string

Name of interface object, specified as a character vector or string.

Example: `'BTdev1'`

Data Types: `char` | `string`

`Timeout` — Time limit for communication
numeric

Time limit in seconds for communication, specified as a numeric value.

Example: `60`

Data Types: `double`

Output Arguments

B — Bluetooth device interface
interface object

Bluetooth device interface, returned as an interface object.

Compatibility Considerations

`Bluetooth` function will be removed
Not recommended starting in R2020b

`Bluetooth` and its object properties will be removed in a future release. Use `bluetooth` (case-sensitive) and its properties instead.

This example shows how to connect to a Bluetooth device using the recommended functionality.

Functionality	Use This Instead
`b = Bluetooth("NXT",3);` `fopen(b)`	`b = bluetooth("NXT",3);`

See “Transition Your Code to bluetooth Interface” for more information about using the recommended functionality.

See Also

Functions
`fclose` | `fopen` | `fread` | `fwrite` | `instrhwinfo`

Introduced in R2011b

clear

Remove instrument objects from MATLAB workspace

Syntax

```
clear obj
```

Arguments

obj	An instrument object or an array of instrument objects.

Description

`clear obj` removes `obj` from the MATLAB workspace.

Examples

This example creates the GPIB object `g`, copies `g` to a new variable `gcopy`, and clears `g` from the MATLAB workspace. `g` is then restored to the workspace with `instrfind` and is shown to be identical to `gcopy`.

```
g = gpib('ni',0,1);
gcopy = g;
clear g
g = instrfind;
isequal(gcopy,g)
ans =
     1
```

Tips

If `obj` is connected to the instrument and it is cleared from the workspace, then `obj` remains connected to the instrument. You can restore `obj` to the workspace with the `instrfind` function. An object connected to the instrument has a `Status` property value of `open`.

To disconnect `obj` from the instrument, use the `fclose` function. To remove `obj` from memory, use the `delete` function. You should remove invalid instrument objects from the workspace with `clear`.

See Also
`delete` | `fclose` | `instrfind` | `instrhelp` | `isvalid` | Status

Introduced before R2006a

clrdevice

(To be removed) Clear instrument buffer

Note This `visa` and `gpib` object function will be removed in a future release. Use `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
clrdevice(obj)
```

Arguments

`obj`	A GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-Serial, or VISA-TCPIP object.

Description

`clrdevice(obj)` clears the hardware buffer of the instrument connected to `obj`.

Tips

Before you can clear the hardware buffer, the instrument must be connected to `obj` with the `fopen` function. A connected object has a `Status` property value of `open`. If you issue `clrdevice` when `obj` is disconnected from the instrument, then an error is returned.

You can clear the software input buffer using the `flushinput` function. You can clear the software output buffer using the `flushoutput` function.

Compatibility Considerations

`visa` object interface will be removed

Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed

Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

`flush` | `flushinput` | `flushoutput` | `fopen` | Status

Introduced before R2006a

commit

Save IVI configuration store object to data file

Syntax

```
commit(obj)
commit(obj, 'file')
```

Arguments

`obj`	IVI configuration store object
`'file'`	Configuration store data file

Description

`commit(obj)` saves the IVI configuration store object, `obj`, to the configuration store data file. The configuration store data file is defined by `obj`'s `ActualLocation` property.

`commit(obj, 'file')` saves the IVI configuration store object, `obj`, to the configuration store data file, `file`. No changes are saved to the configuration store data file that is defined by `obj`'s `ActualLocation` property.

The IVI configuration store object can be modified with the `add`, `update`, and `remove` functions.

See Also

`iviconfigurationstore` | `add` | `remove` | `update`

Introduced before R2006a

configureCallback

Set callback function and trigger condition for communication with serial port device

Syntax

```
configureCallback(device,"terminator",callbackFcn)
configureCallback(device,"byte",count,callbackFcn)
configureCallback(device,"off")
```

Description

`configureCallback(device,"terminator",callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a terminator is available to be read from the specified serial port. The syntax sets the `BytesAvailableFcnMode` property of `device` to `"terminator"` and the `BytesAvailableFcn` property to `callbackFcn`.

Set the terminator character using .

`configureCallback(device,"byte",count,callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a new `count` number of bytes are available to be read. The syntax sets the `BytesAvailableFcnMode` property of `device` to `"byte"`, the `BytesAvailableFcnCount` property to `count`, and the `BytesAvailableFcn` property to `callbackFcn`.

`configureCallback(device,"off")` turns off callbacks. The syntax sets the `BytesAvailableFcnMode` property of `device` to `"off"`.

Examples

Set Serial Port Device Callback and Trigger to "terminator" Mode

Create a connection to a serial port device.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the callback to trigger when a terminator is available to be read.

```
configureCallback(device,"terminator",@callbackFcn)
```

View the properties to confirm the change.

```
device.BytesAvailableFcnMode
device.BytesAvailableFcn
```

```
ans = 

    "terminator"


ans = 

  function_handle with value:

    @callbackFcn
```

Turn the callback off.

```
configureCallback(device,"off")
```

Verify that the callback is off.

```
device.BytesAvailableFcnMode
```

```
ans = 

    "off"
```

Set Serial Port Device Callback and Trigger to "byte" Mode

Create a connection to a serial port device.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the callback to trigger each time 50 new bytes of data are available to be read.

```
configureCallback(device,"byte",50,@callbackFcn)
```

View the properties to confirm the change.

```
device.BytesAvailableFcnMode
device.BytesAvailableFcnCount
device.BytesAvailableFcn
```

```
ans = 
```

```
    "byte"

ans =

    50

ans =

  function_handle with value:

    @callbackFcn
```

Turn the callback off.

```
configureCallback(device,"off")
```

Verify that the callback is off.

```
device.BytesAvailableFcnMode
```

```
ans =

    "off"
```

Read Data from Serial Port Device Using Callback Function

Create a connection to a serial port device.

```
device = serialport("COM3",9600)
```

```
device =

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a callback function that reads ASCII terminated string data and saves it to the `UserData` property of `device`.

```
function readSerialData(src,evt)
    data = readline(src);
    src.UserData = data;
end
```

Set the callback to trigger when a terminator is available to be read.

```
configureCallback(device,"terminator",@readSerialData)
```

Input Arguments

device — Serial port connection
`serialport` object

Serial port connection, specified as a `serialport` object.

Example: `configureCallback(device,"byte",128,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 128 bytes of new data are available to be read from the serial port connection `device`.

`count` — Number of bytes of data to trigger callback
positive integer value

Number of bytes of available data to trigger the callback, specified as a positive integer value. Set the `BytesAvailableFcnCount` property using this argument.

Example: `configureCallback(device,"byte",128,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 128 bytes of new data are available to be read.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`callbackFcn` — Callback function to run when trigger condition is met
function handle

Callback function to run when trigger condition is met, specified as a function handle. The function handle can be a named function handle or an anonymous function with input arguments. Set the `BytesAvailableFcn` property using this argument.

Example: `configureCallback(device,"terminator",@callbackFcn)` sets the `callbackFcn` callback to trigger when a terminator is available to be read.

Data Types: `function_handle`

See Also

Functions
`serialport` | `configureTerminator`

Introduced in R2019b

configureCallback

Set callback function and trigger condition for communication with remote host over TCP/IP

Syntax

```
configureCallback(t,"terminator",callbackFcn)
configureCallback(t,"byte",count,callbackFcn)
configureCallback(t,"off")
```

Description

`configureCallback(t,"terminator",callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a terminator is available to be read from the remote host specified by the TCP/IP client `t`. The syntax sets the `BytesAvailableFcnMode` property of `t` to `"terminator"` and the `BytesAvailableFcn` property to `callbackFcn`.

Set the terminator character using `configureTerminator`.

`configureCallback(t,"byte",count,callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a new `count` number of bytes are available to be read. The syntax sets the `BytesAvailableFcnMode` property of `t` to `"byte"`, the `BytesAvailableFcnCount` property to `count`, and the `BytesAvailableFcn` property to `callbackFcn`.

`configureCallback(t,"off")` turns off callbacks. The syntax sets the `BytesAvailableFcnMode` property of `t` to `"off"`.

Examples

Set Remote Host Callback and Trigger to "terminator" Mode

Create a TCP/IP client called `t`, using the IP address `172.28.154.231` and port `4012`.

```
t = tcpclient("172.28.154.231",4012)
```

```
t = 

  tcpclient with properties:

              Address: '172.28.154.231'
                 Port: 4012
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the callback to trigger when a terminator is available to be read.

```
configureCallback(t,"terminator",@callbackFcn)
```

View the properties to confirm the change.

```
t.BytesAvailableFcnMode
t.BytesAvailableFcn
```

```
ans = 

    "terminator"


ans =

  function_handle with value:

    @callbackFcn
```

Turn the callback off.

```
configureCallback(t,"off")
```

Verify that the callback is off.

```
t.BytesAvailableFcnMode
```

```
ans = 

    "off"
```

Set Remote Host Callback and Trigger to "byte" Mode

Create a TCP/IP client called `t`, using the IP address `172.28.154.231` and port `4012`.

```
t = tcpclient("172.28.154.231",4012)
```

```
t = 

  tcpclient with properties:

              Address: '172.28.154.231'
                 Port: 4012
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the callback to trigger when 50 bytes of data are available to be read.

```
configureCallback(t,"byte",50,@callbackFcn)
```

View the properties to confirm the change.

```
t.BytesAvailableFcnMode
t.BytesAvailableFcnCount
t.BytesAvailableFcn
```

```
ans = 
```

```
    "byte"

ans =

    50

ans =

  function_handle with value:

    @callbackFcn
```

Turn the callback off.

```
configureCallback(t,"off")
```

Verify that the callback is off.

```
t.BytesAvailableFcnMode

ans =

    "off"
```

Input Arguments

t — TCP/IP client
`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `configureCallback(t,"byte",128,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 128 bytes of new data are available to be read from the TCP/IP client `t`.

`count` — Number of bytes of data to trigger callback
positive integer value

Number of bytes of available data to trigger the callback, specified as a positive integer value. Set the `BytesAvailableFcnCount` property using this argument.

Example: `configureCallback(t,"byte",128,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 128 bytes of new data are available to be read.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`callbackFcn` — Callback function to run when trigger condition is met
function handle

Callback function to run when trigger condition is met, specified as a function handle. The function handle can be a named function handle or an anonymous function with input arguments. Set the `BytesAvailableFcn` property using this argument.

Example: `configureCallback(t,"terminator",@callbackFcn)` sets the `callbackFcn` callback to trigger when a terminator is available to be read.

Data Types: `function_handle`

See Also

Functions
`tcpclient` | `configureTerminator`

Introduced in R2020b

configureCallback

Set callback function and trigger condition for communication

Syntax

```
configureCallback(t,"terminator",callbackFcn)
configureCallback(t,"byte",count,callbackFcn)
configureCallback(t,"off")
```

Description

`configureCallback(t,"terminator",callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a terminator is available to be read from the client connected to the TCP/IP server `t`. The syntax sets the `BytesAvailableFcnMode` property of `t` to `"terminator"` and the `BytesAvailableFcn` property to `callbackFcn`.

Set the terminator character using `configureTerminator`.

`configureCallback(t,"byte",count,callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a new `count` number of bytes are available to be read. The syntax sets the `BytesAvailableFcnMode` property of `t` to `"byte"`, the `BytesAvailableFcnCount` property to `count`, and the `BytesAvailableFcn` property to `callbackFcn`.

`configureCallback(t,"off")` turns off callbacks. The syntax sets the `BytesAvailableFcnMode` property of `t` to `"off"`.

Examples

Set TCP/IP Server Callback and Trigger to Terminator Mode

Create a callback function called `readFcn` and save it as a `.m` file in the current working directory.

```
function readFcn(src,~)
message = readline(src);
disp(message)
end
```

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server =
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0
```

```
  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the callback function `readFcn` to trigger when a terminator is available to be read in the server. The callback function `readFcn` reads and displays the message sent from the connected client.

```
configureCallback(server,"terminator",@readFcn)
```

View the properties to confirm the change.

```
server.BytesAvailableFcnMode
```

```
ans = 
"terminator"
```

```
server.BytesAvailableFcn
```

```
ans = function_handle with value:
    @readFcn
```

Write a string of ASCII data to the TCP/IP client. Since the client is connected to the server, this data is available in the server. When the server receives the data with a terminator from the client, `server` triggers its callback function, which displays the data.

```
writeline(client,"Hello, world!")
```

```
Hello, world!
```

Turn the callback off.

```
configureCallback(server,"off")
```

Verify that the callback is off.

```
server.BytesAvailableFcnMode
```

```
ans = 
"off"
```

Set TCP/IP Server Callback and Trigger to Byte Mode

Create a callback function called `readByteFcn` and save it as a `.m` file in the current working directory.

```
function readByteFcn(src,~)
data = read(src,src.NumBytesAvailable);
disp(data)
end
```

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server =
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client =
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the callback function `readByteFcn` to trigger when five bytes of new data are available to be read in the server. The callback function `readByteFcn` reads and displays all the data sent from the connected client.

```
configureCallback(server,"byte",5,@readByteFcn)
```

View the properties to confirm the change.

```
server.BytesAvailableFcnMode
```

```
ans =
"byte"
```

```
server.BytesAvailableFcnCount
```

```
ans = 5
```

```
server.BytesAvailableFcn
```

```
ans = function_handle with value:
    @readByteFcn
```

Write five bytes of data to the TCP/IP client. Since the client is connected to the server, this data is available in the server. When the server receives the five bytes of data from the client, `server` triggers its callback function, which displays the data.

```
write(client,1:5,"uint8")
```

```
1 2 3 4 5
```

Turn the callback off.

```
configureCallback(server,"off")
```

Verify that the callback is off.

```
server.BytesAvailableFcnMode
```

```
ans =
"off"
```

Input Arguments

t — TCP/IP server

`tcpserver` object

TCP/IP server, specified as a `tcpserver` object.

Example: `configureCallback(t,"byte",128,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 128 bytes of new data are available to be read from the TCP/IP server `t`.

count — Number of bytes of data to trigger callback

positive integer value

Number of bytes of available data to trigger the callback, specified as a positive integer value. Set the `BytesAvailableFcnCount` property using this argument.

Example: `configureCallback(t,"byte",128,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 128 bytes of new data are available to be read.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

callbackFcn — Callback function to run when trigger condition is met

function handle

Callback function to run when trigger condition is met, specified as a function handle. The function handle can be a named function handle or an anonymous function with input arguments. Set the `BytesAvailableFcn` property using this argument.

Example: `configureCallback(t,"terminator",@callbackFcn)` sets the `callbackFcn` callback to trigger when a terminator is available to be read.

Data Types: `function_handle`

See Also

`tcpserver` | `configureTerminator`

Introduced in R2021a

configureCallback

Set callback function and trigger condition for communication with UDP socket

Syntax

```
configureCallback(u,"terminator",callbackFcn)
configureCallback(u,"byte",count,callbackFcn)
configureCallback(u,"datagram",count,callbackFcn)
configureCallback(u,"off")
```

Description

`configureCallback(u,"terminator",callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a terminator is available to be read from the specified UDP socket. `u` must be a byte-type `udpport` object. This syntax sets the `BytesAvailableFcnMode` property of `u` to `"terminator"`, and the `BytesAvailableFcn` property to `callbackFcn`.

You set the terminator string with the `configureTerminator` function.

`configureCallback(u,"byte",count,callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a new `count` number of bytes are available to be read. `u` must be a byte-type `udpport` object. This syntax sets the `BytesAvailableFcnMode` property of `u` to `"byte"`, the `BytesAvailableFcnCount` property to `count`, and the `BytesAvailableFcn` property to `callbackFcn`.

`configureCallback(u,"datagram",count,callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a new `count` number of datagrams are available to be read. `u` must be a datagram-type `udpport` object. This syntax sets the `DatagramsAvailableFcnMode` property of `u` to `"datagram"`, the `DatagramsAvailableFcnCount` property to `count`, and the `DatagramsAvailableFcn` property to `callbackFcn`.

`configureCallback(u,"off")` turns off callbacks. This syntax sets the `BytesAvailableFcnMode` or `DatagramsAvailableFcnMode` property of `u` to `"off"`.

Examples

Configure UDP Socket Callback

Create a UDP socket and configure its callback.

Set the callback to trigger when a specified terminator is received.

```
u = udpport;
configureTerminator(u,"CR/LF")
configureCallback(u,"terminator",@myCallback)
```

Set the callback to trigger when 50 bytes are available to read.

```
configureCallback(u,"byte",50,@myCallback)
```

Turn off the callback.

```
configureCallback(u,"off")
```

Input Arguments

u — UDP socket
`udpport` object

UDP socket, specified as a `udpport` object.

Example: `u = udpport`

Data Types: `udpport object`

`count` — Number of bytes or datagrams of data to trigger callback
positive integer value

Number of bytes or datagrams of available data to trigger the callback, specified as a positive integer value. This argument sets the `BytesAvailableFcnCount` or `DatagramsAvailableFcnCount` property.

Example: `configureCallback(u,"byte",128,@readMyData)` sets `readMyData` as the `BytesAvailableFcn` to trigger each time 128 bytes of new data are available to be read.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`callbackFcn` — Callback function to run when trigger condition is met
function handle

Callback function to run when the trigger condition is met, specified as a function handle. The function handle can be a named function handle or an anonymous function with input arguments. This argument sets the `BytesAvailableFcn` property.

Example: `configureCallback(u,"terminator",@readMyData)` sets `readMyData` as the `BytesAvailableFcn` to trigger when a terminator is available.

Data Types: `function_handle`

See Also

Functions
`udpport` | `configureTerminator`

Introduced in R2020b

configureCallback

Set callback function and trigger condition for communication with VISA resource

Syntax

```
configureCallback(v,"terminator",callbackFcn)
configureCallback(v,"byte",count,callbackFcn)
configureCallback(v,"off")
```

Description

`configureCallback(v,"terminator",callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a terminator is available to be read from the VISA resource `v`. The syntax sets the `BytesAvailableFcnMode` property of `v` to `"terminator"` and the `BytesAvailableFcn` property to `callbackFcn`.

Set the terminator character using `configureTerminator`.

`configureCallback(v,"byte",count,callbackFcn)` sets the callback function `callbackFcn` to trigger whenever a new `count` number of bytes are available to be read. The syntax sets the `BytesAvailableFcnMode` property of `v` to `"byte"`, the `BytesAvailableFcnCount` property to `count`, and the `BytesAvailableFcn` property to `callbackFcn`.

`configureCallback(v,"off")` turns off callbacks. The syntax sets the `BytesAvailableFcnMode` property of `v` to `"off"`.

Examples

Set VISA Resource Callback and Trigger to "terminator" Mode

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Set the callback to trigger when a terminator is available to be read.

```
configureCallback(v,"terminator",@callbackFcn)
```

View the properties to confirm the change.

```
v.BytesAvailableFcnMode
v.BytesAvailableFcn
```

```
ans = 

    "terminator"
```

```
ans = 

  function_handle with value:

    @callbackFcn
```

Turn the callback off.

```
configureCallback(v,"off")
```

Verify that the callback is off.

```
v.BytesAvailableFcnMode
```

```
ans = 

    "off"
```

Set VISA Resource Callback and Trigger to "byte" Mode

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Set the callback to trigger when 50 bytes of data are available to be read.

```
configureCallback(v,"byte",50,@callbackFcn)
```

View the properties to confirm the change.

```
v.BytesAvailableFcnMode
v.BytesAvailableFcnCount
v.BytesAvailableFcn
```

```
ans = 

    "byte"


ans =

    50


ans = 

  function_handle with value:

    @callbackFcn
```

Turn the callback off.

```
configureCallback(v,"off")
```

Verify that the callback is off.

```
v.BytesAvailableFcnMode
```

```
ans = 

    "off"
```

Input Arguments

v — VISA resource
visadev object

VISA resource, specified as a `visadev` object.

Example: `configureCallback(v,"byte",128,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 128 bytes of new data are available to be read from the VISA resource `v`.

`count` — Number of bytes of data to trigger callback
positive integer value

Number of bytes of available data to trigger the callback, specified as a positive integer value. Set the `BytesAvailableFcnCount` property using this argument.

Example: `configureCallback(v,"byte",128,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 128 bytes of new data are available to be read.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`callbackFcn` — Callback function to run when trigger condition is met
function handle

Callback function to run when trigger condition is met, specified as a function handle. The function handle can be a named function handle or an anonymous function with input arguments. Set the `BytesAvailableFcn` property using this argument.

Example: `configureCallback(v,"terminator",@callbackFcn)` sets the `callbackFcn` callback to trigger when a terminator is available to be read.

Data Types: `function_handle`

See Also

`visadev` | `configureTerminator`

Introduced in R2021a

configureChannel

Return or set specified oscilloscope control on selected channel

Syntax

```
value = configureChannel(myScope,channel,control)
configureChannel(myScope,channel,Name,Value)
```

Description

`value = configureChannel(myScope,channel,control)` returns the value of the specified oscilloscope control `control` on the selected `channel`.

`configureChannel(myScope,channel,Name,Value)` sets the value of the specified oscilloscope control using a name-value pair argument.

Examples

Set Vertical Range Control of Oscilloscope

Change the vertical range control on an oscilloscope using the Quick-Control Oscilloscope in Instrument Control Toolbox.

Create a connection to the oscilloscope.

```
myScope = oscilloscope("ASRL1::INSTR")
```

```
myScope =
oscilloscope: TEKTRONIX,TDS 1002

   Instrument Settings:
      AcquisitionStartDelay: 'Not supported'
            AcquisitionTime: 2.5 s
               ChannelNames: 'CH1', 'CH2', 'MATH', 'REFA', 'REFB'
            ChannelsEnabled: 'CH1'
            SingleSweepMode: 'off'
                    Timeout: 10 s
             WaveformLength: 2500

   Trigger Settings:
                TriggerMode: 'auto'

   Communication Properties:
                     Status: 'open'
                   Resource: 'ASRL1::INSTR'

lists of methods
```

Get the current vertical range value for `'CH1'`.

```
offset = configureChannel(myScope,'CH1','VerticalRange')
```

```
offset = 5
```

Set the vertical range value to 2.00 V.

```
configureChannel(myScope,'CH1','VerticalRange',2)
```

You can look at the display on the oscilloscope and verify that the vertical offset value has changed to 2.00 V. Get the new vertical range value by reading the value again.

```
offset = configureChannel(myScope,'CH1','VerticalRange')
```

```
offset = 2
```

Input Arguments

`myScope` — Oscilloscope connection

Quick-Control Oscilloscope object

Oscilloscope connection created using `oscilloscope`, specified as a Quick-Control Oscilloscope object.

`channel` — Oscilloscope channel name

character vector | string

Oscilloscope channel name, specified as a character vector or string. See valid channel names by viewing the `ChannelNames` property of your oscilloscope object.

Data Types: `char` | `string`

`control` — Oscilloscope control

`'VerticalCoupling'` | `'VerticalOffset'` | `'VerticalRange'` | `'ProbeAttenuation'`

Oscilloscope control, specified as a character vector or string. Valid values are the same as the parameter names of the “Name-Value Pair Arguments” on page 24-84:

- `'VerticalCoupling'` — Input signal coupling type
- `'VerticalOffset'` — Center of signal range
- `'VerticalRange'` — Input signal range
- `'ProbeAttenuation'` — Probe attenuation

Example: `configureChannel(myScope,'Channel1','VerticalOffset')` returns the current vertical offset value of '`Channel1`' on your oscilloscope.

Data Types: `char` | `string`

Name-Value Pair Arguments

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

Example: `configureChannel(myScope,'Channel1','ProbeAttenuation',10)` sets the probe attenuation value on the channel named `'Channel1'` of the oscilloscope object `myScope` to 10.

`VerticalCoupling` — Input signal coupling type
`'AC'` | `'DC'` | `'GND'`

Input signal coupling type, specified as the comma-separated pair consisting of `'VerticalCoupling'` and one of the following values:

- `'AC'` — Alternating coupling
- `'DC'` — Direct coupling
- `'GND'` — Ground

Example: `configureChannel(myScope,'VerticalCoupling','Channel1','AC')` sets the oscilloscope to apply alternating coupling to the input signal.

Data Types: `char` | `string`

`VerticalOffset` — Center of signal range
numeric

Center of signal range in V, specified as the comma-separated pair consisting of `'VerticalOffset'` and a number.

Example: `configureChannel(myScope,'Channel1','VerticalOffset',5)` sets the oscilloscope to acquire a signal that is centered at 5.0 V.

Data Types: `double`

`VerticalRange` — Input signal range
numeric

Input signal range, specified as the comma-separated pair consisting of `'VerticalRange'` and a number.

Example: `configureChannel(myScope,'Channel1','VerticalRange',2)` sets the oscilloscope to acquire a signal with an input range of 2.00 volts.

Data Types: `double`

`ProbeAttenuation` — Probe attenuation
numeric

Probe attenuation, specified as the comma-separated pair consisting of `'ProbeAttenuation'` and a number. The value must be a multiple of 10.

Example: `configureChannel(myScope,'Channel1','ProbeAttenuation',10)` sets the probe attenuation to 10.

Data Types: `double`

See Also
`oscilloscope` | `readWaveform`

Introduced in R2011b

configureMulticast

Set multicast properties for communication with UDP socket

Syntax

```
configureMulticast(u,address)
configureMulticast(u,address,loopbackEnable)
configureMulticast(u,"off")
```

Description

`configureMulticast(u,address)` subscribes the UDP socket `u` to the multicast group address `address`. This syntax sets the `MulticastGroup` property to `address`, the `EnableMulticastLoopback` property to `true` (the default), and the `EnableMulticast` property to `true`.

This function is only supported on Windows.

`configureMulticast(u,address,loopbackEnable)` subscribes the UDP socket to the specified address, while the `loopbackEnable` property controls whether loopback is enabled. This syntax sets the `EnableMulticastLoopback` accordingly.

`configureMulticast(u,"off")` clears the multicast configuration and related properties.

Examples

Configure UDP Socket Multicast

Create a UDP socket and configure its multicast settings.

Subscribe to a multicast address, allowing loopback.

```
u = udpport;
configureMulticast(u,"226.0.0.1");
```

Subscribe to a multicast address, but ensure that the UDP socket does not receive the data that it sends to the multicast group.

```
configureMulticast(u,"226.0.0.1",false);
```

Unsubscribe from the multicast group.

```
configureMulticast(u,"off");
```

Input Arguments

u — UDP socket
`udpport` object

UDP socket, specified as a `udpport` object.

Example: `u = udpport`

Data Types: `udpport object`

`address` — Multicast group address to subscribe to
string | character vector

Multicast group address to subscribe to, specified as a string or character vector.

Example: `"226.0.0.1"`

Data Types: `char` | `string`

`loopbackEnable` — Allow loopback reading by UDP socket
`true` (default) | `false`

Allow loopback reading by the UDP socket, specified as `false` or `true`. If you specify `true`, the UDP socket can receive messages that it sends to the multicast group, if the `udpport` socket sending the data is itself subscribed to the multicast group.

Example: `false`

Data Types: `logical`

See Also

Functions
`udpport`

Introduced in R2020b

configureTerminator

Set terminator for ASCII string communication with serial port

Syntax

```
configureTerminator(device,terminator)
configureTerminator(device,readterminator,writeterminator)
```

Description

`configureTerminator(device,terminator)` defines the terminator for both read and write communications with the specified serial port. Allowed terminator values are `"LF"` (default), `"CR"`, `"CR/LF"`, and integer values from `0` to `255`. The syntax sets the `Terminator` property of `device`.

After you set the terminator, use `writeline` and `readline` to write and read ASCII terminated string data.

`configureTerminator(device,readterminator,writeterminator)` defines separate terminators for read and write communications.

Examples

Set Same Terminator for Read and Write Communication

Create a connection to a serial port device using `serialport`.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Set both the read and write terminators to `"CR/LF"`.

```
configureTerminator(device,"CR/LF")
```

Confirm the change.

```
device.Terminator
```

```
ans = 
"CR/LF"
```

Set Different Terminators for Read and Write Communication

Create a connection to a serial port device using `serialport`.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the read terminator to `"CR"` and the write terminator to `10`.

```
configureTerminator(device,"CR",10)
```

Confirm the change.

```
device.Terminator
```

```
ans=1×2 cell array
    {["CR"]}    {[10]}
```

The first element in the array is the read terminator and the second is the write terminator.

Write and Read Line of ASCII Data from Serial Port Device

Create a connection to a serial port device. In this example, the serial port at `COM3` is connected to a loopback device.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Check the default ASCII terminator.

```
device.Terminator
```

```
ans = 

    "LF"
```

Set the terminator to `"CR"` and write a string of ASCII data. The `writeline` function automatically appends the terminator to the data.

```
configureTerminator(device,"CR")
writeline(device,"hello")
```

Write another string of ASCII data with the terminator automatically appended.

```
writeline(device,"world")
```

Since the port is connected to a loopback device, the data you write to the device is returned to MATLAB. Read a string of ASCII data. The `readline` function returns data until it reaches a terminator.

```
readline(device)
```

```
ans = 

    "hello"
```

Read a string of ASCII data again to return the second string that you wrote.

```
readline(device)
```

```
ans = 

    "world"
```

Clear the serial port connection.

```
clear device
```

Input Arguments

device — Serial port connection
`serialport` object

Serial port connection, specified as a `serialport` object.

Example: `configureTerminator(device,"CR")` sets the terminator on the serial port connection `device`.

terminator — ASCII terminator
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

ASCII terminator for read and write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from 0 to 255. Use this form when setting the same terminator for both read and write. When reading from the serial port with a terminator value of `"CR/LF"`, the read terminates on the occurrence of `CR` and `LF` together. When writing to the serial port with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property.

Example: `configureTerminator(device,"CR/LF")` sets both the read and write terminators to `"CR/LF"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`readterminator,writeterminator` — ASCII terminators for read and write
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

ASCII terminators for read or write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from 0 to 255. Use this form when setting different terminators for read and write. When reading from the serial port with a terminator value of `"CR/LF"`, the read terminates on the occurrence of `CR` and `LF` together. When writing to the serial port with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property.

Example: `configureTerminator(device,"CR",10)` sets the read terminator to `"CR"` and write terminator to `10`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

See Also

Functions
`writeline` | `readline` | `serialport` | `configureCallback`

Introduced in R2019b

configureTerminator

Set terminator for ASCII string communication with remote host over TCP/IP

Syntax

```
configureTerminator(t,terminator)
configureTerminator(t,readterminator,writeterminator)
```

Description

`configureTerminator(t,terminator)` defines the terminator for both read and write communications with the remote host specified by the TCP/IP client `t`. Allowed terminator values are `"LF"` (default), `"CR"`, `"CR/LF"`, and integer values from `0` to `255`. The syntax sets the `Terminator` property of `t`.

After you set the terminator, use `writeline` and `readline` to write and read ASCII terminated string data.

`configureTerminator(t,readterminator,writeterminator)` defines separate terminators for read and write communications.

Examples

Set Same Terminator for Read and Write Communication

Create a TCP/IP client called `t`, using the IP address `172.28.154.231` and port `4012`.

```
t = tcpclient("172.28.154.231",4012)
```

```
t = 

  tcpclient with properties:

              Address: '172.28.154.231'
                 Port: 4012
    NumBytesAvailable: 0

  Show all properties, functions
```

Set both the read and write terminators to `"CR/LF"`.

```
configureTerminator(t,"CR/LF")
```

Confirm the change.

```
t.Terminator
```

```
ans = 
"CR/LF"
```

Set Different Terminators for Read and Write Communication

Create a TCP/IP client called `t`, using the IP address `172.28.154.231` and port `4012`.

```
t = tcpclient("172.28.154.231",4012)
```

```
t = 

  tcpclient with properties:

              Address: '172.28.154.231'
                 Port: 4012
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the read terminator to `"CR"` and the write terminator to `10`.

```
configureTerminator(t,"CR",10)
```

Confirm the change.

```
t.Terminator
```

```
ans=1×2 cell array
    {["CR"]}    {[10]}
```

The first element in the array is the read terminator and the second is the write terminator.

Write and Read Line of ASCII Data from Remote Host

Create a TCP/IP client connection called `t`, connecting to a TCP/IP echo server with port 4000. To do so, you must have an `echotcpip` server running on port 4000.

```
echotcpip("on",4000)
t = tcpclient("localhost",4000)
```

```
t = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Check the default ASCII terminator.

```
t.Terminator

ans = 
"LF"
```

Set the terminator to `"CR"` and write a string of ASCII data. The `writeline` function automatically appends the terminator to the data.

```
configureTerminator(t,"CR")
writeline(t,"hello")
```

Write another string of ASCII data with the terminator automatically appended.

```
writeline(t,"world")
```

Since the client is connected to an echo server, the data you write to the server is returned to the client. Read a string of ASCII data. The `readline` function returns data until it reaches a terminator.

```
readline(t)

ans = 
"hello"
```

Read a string of ASCII data again to return the second string that you wrote.

```
readline(t)

ans = 
"world"
```

Close the echo server and clear the TCP/IP client connection.

```
echotcpip("off")
clear t
```

Input Arguments

t — TCP/IP client

`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `configureTerminator(t,"CR/LF")` sets the terminator value for the TCP/IP client `t`.

`terminator` — ASCII terminator

`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

ASCII terminator for read and write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from 0 to 255. Use this form when setting the same terminator for both read and write. When reading from the remote host with a terminator value of `"CR/LF"`, the read terminates on the occurrence of `CR` and `LF` together. When writing to the remote host with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property.

Example: `configureTerminator(t,"CR/LF")` sets both the read and write terminators to `"CR/LF"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`readterminator,writeterminator` — ASCII terminators for read and write
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

ASCII terminators for read or write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from 0 to 255. Use this form when setting different terminators for read and write. When reading from the remote host with a terminator value of `"CR/LF"`, the read terminates on the occurrence of `CR` and `LF` together. When writing to the remote host with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property.

Example: `configureTerminator(t,"CR",10)` sets the read terminator to `"CR"` and write terminator to `10`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

See Also

Functions
`tcpclient` | `configureCallback` | `readline` | `writeline`

Introduced in R2020b

configureTerminator

Set terminator for ASCII string communication

Syntax

```
configureTerminator(t,terminator)
configureTerminator(t,readterminator,writeterminator)
```

Description

`configureTerminator(t,terminator)` defines the terminator for both read and write communications with the client connected to the TCP/IP server `t`. Allowed terminator values are `"LF"` (default), `"CR"`, `"CR/LF"`, and integer values from 0 to 255. The syntax sets the `Terminator` property of `t`.

After you set the terminator, use `writeline` and `readline` to write and read ASCII-terminated string data.

`configureTerminator(t,readterminator,writeterminator)` defines separate terminators for read and write communications.

Examples

Set Same Terminator for Read and Write Communication

Create a TCP/IP server on port 4000.

```
server = tcpserver(4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "::"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Set both the read and write terminators to `"CR/LF"`.

```
configureTerminator(server,"CR/LF")
```

If you have a client connected to the server, you must set the same terminators for the client and server to successfully perform `readline` and `writeline` operations.

Confirm the change.

```
server.Terminator
```

```
ans =
"CR/LF"
```

Set Different Terminators for Read and Write Communication

Create a TCP/IP server on port 4000.

```
server = tcpserver(4000)
```

```
server =
  TCPServer with properties:

        ServerAddress: "::"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Set the read terminator to `"CR"` and the write terminator to `10`.

```
configureTerminator(server,"CR",10)
```

If you have a client connected to the server, you must set the same terminators for the client and server to successfully perform `readline` and `writeline` operations.

Confirm the change.

```
server.Terminator
```

```
ans=1×2 cell array
    {["CR"]}    {[10]}
```

Write Line of ASCII Data from TCP/IP Server

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server =
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
```

```
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client =
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Check the default ASCII terminator for the server.

```
server.Terminator
```

```
ans =
"LF"
```

Set the terminators for both the server and client to `"CR"`. The TCP/IP server and its connected client must have the same terminator.

```
configureTerminator(server,"CR")
configureTerminator(client,"CR")
```

Write a string of ASCII data from the server to the client by writing it to the `server` object. The `writeline` function automatically appends the terminator to the data.

```
writeline(server,"hello")
```

Write another string of ASCII data with the terminator automatically appended.

```
writeline(server,"world")
```

Since the client is connected to the server, the data you write is available in the client. Read a string of ASCII data from the `client` object. The `readline` function returns data until it reaches a terminator.

```
readline(client)
```

```
ans =
"hello"
```

Read a string of ASCII data again to return the second string.

```
readline(client)
```

```
ans =
"world"
```

Input Arguments

`t` — TCP/IP server
`tcpserver` object

TCP/IP server, specified as a `tcpserver` object.

Example: `configureTerminator(t,"CR/LF")` sets the terminator value for the TCP/IP server `t`.

`terminator` — ASCII terminator
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

ASCII terminator for read and write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from 0 to 255. Use this form when setting the same terminator for both read and write. When reading from the remote host with a terminator value of `"CR/LF"`, the read terminates on an occurrence of `CR` and `LF` together. When writing to the remote host with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property.

Example: `configureTerminator(t,"CR/LF")` sets both the read and write terminators to `"CR/LF"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`readterminator,writeterminator` — ASCII terminators for read and write
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

ASCII terminators for read or write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from 0 to 255. Use this form when setting different terminators for read and write. When reading from the remote host with a terminator value of `"CR/LF"`, the read terminates on an occurrence of `CR` and `LF` together. When writing to the remote host with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property.

Example: `configureTerminator(t,"CR",10)` sets the read terminator to `"CR"` and write terminator to `10`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

See Also

`tcpserver` | `configureCallback` | `readline` | `writeline`

Introduced in R2021a

configureTerminator

Set terminator for ASCII string communication with UDP socket

Syntax

```
configureTerminator(u,terminator)
configureTerminator(u,readterminator,writeterminator)
```

Description

`configureTerminator(u,terminator)` defines the terminator for both read and write communications with the specified UDP socket. `u` must be a byte-type `udpport` object. Allowed terminator values are `"LF"` (default), `"CR"`, `"CR/LF"`, and integer values from `0` to `255`. The syntax sets the `Terminator` property of `u`.

After you set the terminator, use `writeline` and `readline` to write and read ASCII-terminated string data.

`configureTerminator(u,readterminator,writeterminator)` defines separate terminators for read and write communications.

Examples

Set Common Terminator for Read and Write Communication

Create a UDP socket and set its read and write terminators to `"CR/LF"`.

```
u = udpport;
configureTerminator(u,"CR/LF")
```

Confirm the change.

```
u.Terminator
```

```
ans = 

    "CR/LF"
```

Set Different Terminators for Read and Write Communication

Create a UDP socket and set its read terminator to `"CR"` and its write terminator to `10`.

```
u = udpport;
configureTerminator(u,"CR",10)
```

Confirm the change.

```
u.Terminator
```

```
ans =

  1×2 cell array

    {["CR"]}    {[10]}
```

The first element in the array is the read terminator and the second is the write terminator.

Input Arguments

u — Byte-type UDP socket
`udpport` object

Byte-type UDP socket, specified as a `udpport` object.

Example: `u = udpport`

Data Types: `udpport object`

`terminator` — ASCII terminator
`"LF"` (default) | `"CR"` | `"CR/LF"` | `0` to `255`

ASCII terminator for read and write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from `0` to `255`. Use this form when setting the same terminator for both read and write. When you read from the UDP socket with a terminator value of `"CR/LF"`, the read terminates on the occurrence of `CR` and `LF` together. When you write to the UDP socket with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property.

Example: `configureTerminator(u,"CR/LF")` sets both the read and write terminators to `"CR/LF"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`readterminator,writeterminator` — ASCII terminators for read and write
`"LF"` (default) | `"CR"` | `"CR/LF"` | `0` to `255`

ASCII terminators for read or write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from `0` to `255`. Use this form when setting different terminators for read and write. When you read from the UDP socket with a terminator value of `"CR/LF"`, the read terminates on the occurrence of `CR` and `LF` together. When you write to the UDP socket with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property to a cell array of `{readterminator,writeterminator}`.

Example: `configureTerminator(u,"CR",10)` sets the read terminator to `"CR"` and the write terminator to `10`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

See Also

Functions
`udpport` | `writeline` | `readline`

Introduced in R2020b

configureTerminator

Set terminator for ASCII string communication with VISA resource

Syntax

```
configureTerminator(v,terminator)
configureTerminator(v,readterminator,writeterminator)
```

Description

`configureTerminator(v,terminator)` defines the terminator for both read and write communications with the VISA resource `v`. Allowed terminator values are `"LF"` (default), `"CR"`, `"CR/LF"`, and integer values from 0 to 255. The syntax sets the `Terminator` property of `v`.

After you set the terminator, use `writeline` and `readline` to write and read ASCII-terminated string data.

`configureTerminator(v,readterminator,writeterminator)` defines separate terminators for read and write communications.

Examples

Set Same Terminator for Read and Write Communication

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Set both the read and write terminators to `"CR/LF"`.

```
configureTerminator(v,"CR/LF")
```

Confirm the change.

```
v.Terminator
```

```
ans = 
"CR/LF"
```

Set Different Terminators for Read and Write Communication

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Set the read terminator to `"CR"` and the write terminator to `10`.

```
configureTerminator(v,"CR",10)
```

Confirm the change.

```
v.Terminator
```

```
ans=1×2 cell array
    {["CR"]}    {[10]}
```

The first element in the array is the read terminator and the second is the write terminator.

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `configureTerminator(v,"CR/LF")` sets the terminator value for the VISA resource `v`.

`terminator` — ASCII terminator
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

ASCII terminator for read and write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from 0 to 255. Use this form when setting the same terminator for both read and write. When reading from the remote host with a terminator value of `"CR/LF"`, the read terminates on an occurrence of `CR` and `LF` together. When writing to the remote host with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property.

Example: `configureTerminator(v,"CR/LF")` sets both the read and write terminators to `"CR/LF"`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`readterminator,writeterminator` — ASCII terminators for read and write
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

ASCII terminators for read or write communication, specified as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer value from 0 to 255. Use this form when setting different terminators for read and write. When reading from the remote host with a terminator value of `"CR/LF"`, the read terminates on an occurrence of `CR` and `LF` together. When writing to the remote host with a terminator value of `"CR/LF"`, the write terminates by adding both `CR` and `LF`. This input argument sets the `Terminator` property.

Example: `configureTerminator(v,"CR",10)` sets the read terminator to `"CR"` and write terminator to `10`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

See Also

`visadev` | `configureCallback` | `readline` | `writeline`

Introduced in R2021a

connect

Connect device object to instrument

Syntax

```
connect(obj)
connect(obj,'update')
```

Arguments

`obj`	A device object or an array of device objects.
`update`	Update the state of the object or the instrument.

Description

`connect(obj)` connects the device object specified by `obj` to the instrument. `obj` can be an array of device objects.

`connect(obj,'update')` updates the state of the object or the instrument. `update` can be `object` or `instrument`. If `update` is `object`, the object is updated to reflect the state of the instrument. If `update` is `instrument`, the instrument is updated to reflect the state of the object. In this case, all property values defined by the object are sent to the instrument on open. By default, `update` is `object`.

Examples

Create a device object for a Tektronix TDS 210 oscilloscope that is connected to a National Instruments GPIB board.

```
g = gpib('ni',0,2);
d = icdevice('tektronix_tds210',g);
```

Connect to the instrument.

```
connect(d)
```

List the oscilloscope settings that can be configured.

```
props = set(d);
```

Get the current configuration of the oscilloscope.

```
values = get(d);
```

Disconnect from the instrument and clean up.

```
disconnect(d)
delete([d g])
```

Tips

If `obj` is successfully connected to the instrument, its `Status` property is configured to `open`. If `obj` is an array of device objects and one of the objects cannot be connected to the instrument, the remaining objects in the array will be connected and a warning is displayed.

See Also

`disconnect` | `delete` | `instrhelp` | Status

Introduced before R2006a

delete

Remove instrument objects from memory

Syntax

`delete(obj)`

Arguments

`obj`	An instrument object or an array of instrument objects.

Description

`delete(obj)` removes `obj` from memory.

Examples

This example creates the GPIB object `g`, connects `g` to the instrument, writes and reads text data, disconnects `g`, removes `g` from memory using `delete`, and then removes `g` from the workspace using `clear`.

```
g = gpib('ni',0,1);
fopen(g)
fprintf(g,'*IDN?')
idn = fscanf(g);
fclose(g)
delete(g)
clear g
```

Tips

When you delete `obj`, it becomes an *invalid* object. Because you cannot connect an invalid object to the instrument, you should remove it from the workspace with the `clear` command. If multiple references to `obj` exist in the workspace, then deleting one reference invalidates the remaining references.

If `obj` is connected to the instrument, it has a `Status` property value of `open`. If you issue `delete` while `obj` is connected, the connection is automatically broken. You can also disconnect `obj` from the instrument with the `fclose` function.

If `obj` is an interface object that is associated with a device object, the device object is automatically deleted when `obj` is deleted. However, if `obj` is a device object, the interface object is not automatically deleted when `obj` is deleted.

See Also

`clear` | `fclose` | `instrhelp` | `isvalid` | `stopasync` | Status

Introduced before R2006a

devicereset

Reset instrument

Syntax

```
devicereset(obj)
```

Arguments

`obj`	A device object.

Description

`devicereset(obj)` resets the instrument associated with the device object specified by `obj`.

Introduced before R2006a

disconnect

Disconnect device object from instrument

Syntax

`disconnect(obj)`

Arguments

`obj`	A device object or an array of device objects.

Description

`disconnect(obj)` disconnects the device object specified by `obj` from the instrument.

Examples

Create a device object for a Tektronix TDS 210 oscilloscope that is connected to a National Instruments GPIB board.

```
g = gpib('ni',0,2);
d = icdevice('tektronix_tds210',g);
```

Connect to the instrument.

```
connect(d)
```

Get the current configuration of the oscilloscope.

```
values = get(d);
```

Disconnect from the instrument and clean up.

```
disconnect(d)
delete([d g])
```

Tips

If `obj` is disconnected from the instrument, its `Status` property is configured to `closed`. You can reconnect to the instrument with the `connect` function. If `obj` is an array of device objects and one of the objects cannot be disconnected from the instrument, the remaining objects in the array will be disconnected and a warning is displayed.

See Also

`connect` | `delete` | `instrhelp` | Status

Introduced before R2006a

disp

Display instrument object summary information

Syntax

```
obj
disp(obj)
```

Arguments

`obj`	An instrument object or an array of instrument objects.

Description

`obj` or `disp(obj)` displays summary information for `obj`.

Examples

The following commands display summary information for the GPIB object `g`.

```
g = gpib('ni',0,1)
g.EOSMode = 'read'
g
```

Tips

In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when

- Creating an instrument object
- Configuring property values using the dot notation

You can also display summary information via the Workspace browser by right-clicking an instrument object, and selecting **Display Summary** from the context menu.

Introduced before R2006a

download

Downloads arbitrary waveform to RF signal generator

Syntax

`download(rf, IQData, SampleRate)`

Description

`download(rf, IQData, SampleRate)` downloads an arbitrary waveform to the RF signal generator, `rf`. It accepts a complex vector of doubles containing the `IQData` and a double defining the `SampleRate` of the signal.

Examples

Download a Waveform to RF Signal Generator

You can download a waveform to your `rfsiggen` object and assign the `IQData` and `SampleRate` to use.

Create an `rfsiggen` object to communicate with an RF signal generator using the VISA resource string and driver associated with your own instrument.

```
rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRfSigGen')
```

When you designate the `Resource` and `Driver` properties during object creation, it automatically connects to the instrument.

Assign the `IQData` and `SampleRate` variables to use in the download.

```
IQData = (-0.98:0.02:1) + 1i*(-0.98:0.02:1);
SampleRate = 800000;
```

Perform the download.

```
download(rf, IQData, SampleRate)
```

Input Arguments

`IQData` — IQ data to use in the download

vector of doubles

IQ data to use in the download, specified as a vector of doubles. This example downloads the data to the RF Signal Generator object `rf` using the specified IQ data and sample rate.

Example: `download(rf, IQData, SampleRate)`

Data Types: `double`
Complex Number Support: Yes

`SampleRate` — Sample rate of the signal
double

Sample rate of the signal, specified as a double. This example downloads the data to the RF signal generator object `rf` using the specified IQ data and sample rate.

Example: `download(rf, IQData, SampleRate)`

Data Types: `double`

See Also

`rfsiggen` | `start` | `resources` | `drivers`

Topics

"Download and Generate Signals with RF Signal Generator" on page 14-45
"Quick-Control RF Signal Generator Functions" on page 14-41
"Quick-Control RF Signal Generator Properties" on page 14-43

Introduced in R2017b

drivers

List of available instrument drivers for Quick-Control interfaces

Syntax

```
drivers(rf)
```

Description

`drivers(rf)` lists the drivers for RF signal generator object `rf`. It returns a list of drivers for the Quick-Control RF Signal Generator, Quick-Control Oscilloscope, or Quick-Control Function Generator objects. It also lists instrument model information for each driver.

Examples

List Drivers and Connect to RF Signal Generator

The `drivers` function can list drivers available for any of the Quick-Control interface objects: RF signal generator (`rfsiggen`), oscilloscope (`oscilloscope`), or function generator (`fgen`). This example uses Quick-Control RF Signal Generator, but the function also works in the same way for the other two object types.

Create an RF signal generator object without assigning the resource or driver.

```
rf = rfsiggen;
```

List the drivers.

```
drivers(rf)
```

```
ans =

     Driver: AgRfSigGen_SCPI
     Supported Models:
     E4428C, E4438C

     Driver: RsRfSigGen_SCPI
     Supported Models:
     SMW200A, SMBV100A, SMU200A, SMJ100A, AMU200A, SMATE200A

     Driver: AgRfSigGen
     Supported Models:
     E4428C,E4438C,N5181A,N5182A,N5183A,N5171B,N5181B,N5172B
     N5182B,N5173B,N5183B,E8241A,E8244A,E8251A,E8254A,E8247C
```

In this case, it finds the drivers for a Keysight (formerly Agilent) SCPI-based RF signal generator, a Rohde & Shwartz SCPI-based generator, and another Keysight generator. You can see that it lists supported models in each case.

Set the RF signal generator resource using the `Resource` property, which is the VISA resource string.

```
rf.Resource = 'TCPIP0::172.28.22.99::inst0::INSTR';
```

Set the RF signal generator driver using the `Driver` property. The driver name came from using the `drivers` function in step 2.

```
rf.Driver = 'AgRfSigGen';
```

You can now connect to the instrument.

```
connect(rf);
```

See Also

`rfsiggen` | `download` | `start` | `resources`

Topics

“Download and Generate Signals with RF Signal Generator” on page 14-45
“Quick-Control RF Signal Generator Functions” on page 14-41
“Quick-Control RF Signal Generator Properties” on page 14-43

Introduced in R2017b

echotcpip

Start or stop TCP/IP echo server

Syntax

```
echotcpip("on",port)
echotcpip("off")
```

Description

`echotcpip("on",port)` starts a TCP/IP server at the specified port number.

`echotcpip("off")` stops the echo server.

Examples

Communicate with Echo TCP/IP Server

This example shows how to set up an echo TCP/IP server. Start the echo server on port 4000 and create a TCP/IP client object. Connect the TCP/IP client object to the host.

```
echotcpip('on',4000)
t = tcpclient('localhost',4000);
```

Write to the host and read from the host.

```
writeline(t,'echo this string.')
data = readline(t)
```

```
data = 

    "echo this string."
```

Stop the echo server, disconnect the TCP/IP client object from the host, and clear the TCP/IP client object.

```
echotcpip('off')
clear t
```

Input Arguments

`port` — Port number of the server
numeric

Port number of the server, specified as a number between 1 and 65535, inclusive.

Data Types: `double`

See Also

Functions
`echoudp` | `udpport` | `tcpclient`

Topics
"Communicate Binary and ASCII Data to an Echo Server Using TCP/IP" on page 22-124

Introduced before R2006a

echoudp

Start or stop echo UDP server

Syntax

```
echoudp("on",port)
echoudp("off")
```

Description

`echoudp("on",port)` starts a UDP server at the specified port number.

`echoudp("off")` stops the echo server.

Examples

Communicate with an Echo UDP Server

This example shows how to set up an echo UDP server.

Start the echoserver using the `echoudp` function on port 4012. Create a `udpport` instance to communicate with the echo server.

```
echoudp("on",4012)
u = udpport
```

```
u = 
  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 58379
    NumBytesAvailable: 0

  Show all properties, functions
```

Write data using the `write` function to the echo server running on port 4012.

```
write(u,1:10,"uint8","127.0.0.1",4012);
```

The echo server sends back the data written to it. Read this data back using the `read` function.

```
data = read(u,10)
```

```
data = 1×10

     1     2     3     4     5     6     7     8     9    10
```

Stop the echo server and clear the `udpport` object.

```
echoudp("off")
clear u
```

Input Arguments

port — Port number of the server
numeric

Port number of the server, specified as a number between 1 and 65535, inclusive.

Data Types: double

See Also

udpport | tcpclient | echotcpip

Introduced before R2006a

fclose

Disconnect interface object from instrument

Syntax

`fclose(obj)`

Arguments

`obj`	An interface object or an array of interface objects.

Description

`fclose(obj)` disconnects `obj` from the instrument.

Examples

This example creates the GPIB object `g`, connects `g` to the instrument, writes and reads text data, and then disconnects `g` from the instrument using `fclose`.

```
g = gpib('ni',0,1);
fopen(g)
fprintf(g,'*IDN?')
idn = fscanf(g);
fclose(g)
```

At this point, you can once again connect an interface object to the instrument. If you no longer need `g`, you should remove it from memory with the `delete` function, and remove it from the workspace with the `clear` command.

Tips

If `obj` was successfully disconnected, then the `Status` property is configured to `closed` and the `RecordStatus` property is configured to `off`. You can reconnect `obj` to the instrument using the `fopen` function.

An error is returned if you issue `fclose` while data is being written asynchronously. In this case, you should abort the write operation with the `stopasync` function, or wait for the write operation to complete.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See “Transition Your Code to bluetooth Interface” for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See “Transition Your Code to tcpclient Interface” on page 7-36 or “Transition Your Code to tcpserver Interface” on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See “Transition Your Code to udpport Interface” on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`clear` | `delete` | `fopen` | `instrhelp` | `record` | `stopasync`

Properties
RecordStatus | Status

Introduced before R2006a

fgen

Create Quick-Control Function Generator object

Syntax

```
myFGen = fgen()
connect(myFGen);
set(myFGen, 'P1',V1,'P2',V2,...)
enableOutput(myFGen);
```

Description

The Quick-Control Function Generator can be used for any function generator that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. This fgen object is easy to use.

`myFGen = fgen()` creates an instance of the function generator named `myFGen`.

`connect(myFGen);` connects to the function generator.

`set(myFGen, 'P1',V1,'P2',V2,...)` assigns the specified property values.

`enableOutput(myFGen);` enables the function generator to produce a signal that appears at the output connector.

For information on the prerequisites for using `fgen`, see “Quick-Control Function Generator Requirements” on page 14-30.

The Quick-Control Function Generator `fgen` function can use the following special functions, in addition to standard functions such as `connect` and `disconnect`.

Function	Description
`selectChannel`	Specifies the channel name from which the function generator produces the waveform. Example: `selectChannel(myFGen, '1');`
`drivers`	Returns a list of available function generator instrument drivers. Example: `driverlist = drivers(myFGen);`
`resources`	Retrieves a list of available instrument resources. It returns a list of available VISA resource strings when using an IVI-C function generator. Example: `res = resources(myFGen);`

Function	Description
`selectWaveform`	Specifies which arbitrary waveform the function generator produces. Example: `selectWaveform (myFGen, wh);` where `wh` is the waveform handle you are selecting.

`downloadWaveform`	Downloads an arbitrary waveform to the function generator. If you provide an output variable, a waveform handle is returned. It can be used in the `selectWaveform` and `removeWaveform` functions.
	
	If you don't provide an output variable, function generator will overwrite the waveform when a new waveform is downloaded and deletes it upon disconnection.
	
	Example:
	
	`% To download the following waveform to fgen`
	`w = 1:0.001:2;`
	`downloadWaveform (myFGen, w);`
	
	`% To download a waveform to fgen and return a waveform handle`
	`wh = downloadWaveform (myFGen, w);`

`removeWaveform`	Removes a previously created arbitrary waveform from the function generator's memory. If a waveform handle is provided, it removes the waveform represented by the waveform handle.
	
	Example:
	
	`% Remove a waveform from fgen with waveform handle 10000`
	`removeWaveform (myFGen, 10000);`

`enableOutput`	Enables the function generator to produce a signal that appears at the output connector. This function produces a waveform defined by the `Waveform` property. If the `Waveform` property is set to `'Arb'`, the function uses the latest internal waveform handle to output the waveform. `enableOutput (myFGen);`
`disableOutput`	Disables the signal that appears at the output connector. Disables the selected channel. `disableOutput (myFGen);`
`reset`	Sets the function generator to factory state.

Arguments

The Quick-Control Function Generator `fgen` can use the following properties.

Property	Description
AMDepth	Specifies the extent of Amplitude modulation the function generator applies to the carrier signal. The units are a percentage of full modulation. At 0% depth, the output amplitude equals the carrier signal's amplitude. At 100% depth, the output amplitude equals twice the carrier signal's amplitude. This property affects function generator behavior only when the Mode is set to 'AM' and ModulationResource is set to 'internal'.
Amplitude	Specifies the amplitude of the standard waveform. The value is the amplitude at the output terminal. The units are volts peak-to-peak (Vpp). For example, to produce a waveform ranging from -5.0 to +5.0 volts, set this value to 10.0 volts. Does not apply if Waveform is of type 'Arb'.
ArbWaveformGain	Specifies the factor by which the function generator scales the arbitrary waveform data. Use this property to scale the arbitrary waveform to ranges other than -1.0 to +1.0. When set to 2.0, the output signal ranges from -2.0 to +2.0 volts. Only applies if Waveform is of type 'Arb'.
BurstCount	Specifies the number of waveform cycles that the function generator produces after it receives a trigger. Only applies if Mode is set to 'burst'.
ChannelNames	This read-only property provides available channel names in a cell array.
Driver	This property is optional. Use only if necessary to specify the underlying driver used to communicate with an instrument. If the DriverDetectionMode property is set to 'manual', use the Driver property to specify the instrument driver.
DriverDetectionMode	Sets up criteria for connection. Valid values are 'auto' and 'manual'. The default value is 'auto', which means you do not need to set a driver name before connecting to an instrument. If set to 'manual', a driver name needs to be provided using the Driver property before connecting to instrument.
FMDeviation	Specifies the maximum frequency deviation the modulating waveform applies to the carrier waveform. This deviation corresponds to the maximum amplitude level of the modulating signal. The units are Hertz (Hz). This property affects function generator behavior only when Mode is set to 'FM' and ModulationSource is set to 'internal'.
Frequency	Specifies the rate at which the function generator outputs an entire arbitrary waveform when Waveform is set to 'Arb'. It specifies the frequency of the standard waveform when Waveform is set to standard waveform types. The units are Hertz (Hz).

Property	Description
Mode	Specifies run mode. Valid values are 'continuous', 'burst', 'AM', or 'FM'. Specifies how the function generator produces waveforms. It configures the instrument to generate output continuously or to generate a discrete number of waveform cycles based on a trigger event. It can also be set to AM and FM.
ModulationFrequency	Specifies the frequency of the standard waveform that the function generator uses to modulate the output signal. The units are Hertz (Hz). This attribute affects function generator behavior only when Mode is set to 'AM' or 'FM' and the ModulationSource attribute is set to 'internal'.
ModulationSource	Specifies the signal that the function generator uses to modulate the output signal. Valid values are 'internal' and 'external'. This attribute affects function generator behavior only when Mode is set to 'AM' or 'FM'.
ModulationWaveform	Specifies the standard waveform type that the function generator uses to modulate the output signal. This affects function generator behavior only when Mode is set to 'AM' or 'FM' and the ModulationSource is set to 'internal'. Valid values are 'sine', 'square', 'triangle', 'RampUp', 'RampDown', and 'DC'.
Offset	Uses the standard waveform DC offset as input arguments if the waveform is not of type 'Arb'. Use Arb Waveform Offset as input arguments if the waveform is of type 'Arb'. Specifies the DC offset of the standard waveform when Waveform is set to standard waveform. For example, a standard waveform ranging from +5.0 volts to 0.0 volts has a DC offset of 2.5 volts. When Waveform is set to 'Arb', this property shifts the arbitrary waveform's range. For example, when it is set to 1.0, the output signal ranges from 2.0 volts to 0.0 volts.
OutputImpedance	Specifies the function generator's output impedance at the output connector.
Resource	Set this before connecting to the instrument. It is the VISA resource string for your instrument.
SelectedChannel	Returns the selected channel name that was set using the selectChannel function.
StartPhase	Specifies the horizontal offset in degrees of the standard waveform the function generator produces. The units are degrees of one waveform cycle. For example, a 180-degree phase offset means output generation begins halfway through the waveform.
Status	This read-only property indicates the communication status of your instrument session. It is either 'open' or 'closed'.

Property	Description
TriggerRate	Specifies the rate at which the function generator's internal trigger source produces a trigger, in triggers per second. This property affects function generator behavior only when the TriggerSource is set to 'internal'. Only applies if Mode is set to 'burst'.
TriggerSource	Specifies the trigger source. After the function generator receives a trigger, it generates an output signal if Mode is set to 'burst'. Valid values are 'internal' or 'external'.
Waveform	Uses the waveform type as an input argument. Valid values are 'Arb', for an arbitrary waveform, or these standard waveform types – 'Sine', 'Square', 'Triangle', 'RampUp', 'RampDown', and 'DC'.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Examples

Create an instance of the function generator called myFGen.

```
myFGen = fgen()
```

Discover available resources. A resource string is an identifier to the instrument. You need to set it before connecting to the instrument.

```
availableResources = resources(myFGen)
```

Set the resource. In this example, we are controlling an instrument that is connected via GPIB with a board index of 0 and primary address of 10.

```
myFGen.Resource = 'GPIB0::10::INSTR';
```

Connect to the function generator.

```
connect(myFGen);
```

Specify the channel name from which the function generator produces the waveform.

```
selectChannel(myFGen, '1');
```

Configure a standard waveform to be a continuous sine wave.

```
set(myFGen, 'Waveform', 'sine');
set(myFGen, 'Mode', 'continuous');
```

Configure the function generator.

```
% Set the load impedance to 50 Ohms.
set(myFGen, 'OutputImpedance', 50);
```

```
% Set the frequency to 2500 Hz.
set(myFGen, 'Frequency', 2500);

% Set the amplitude to 1.2 volts.
set(myFGen, 'Amplitude', 1.2);

% Set the offset to 0.4 volts.
set(myFGen, 'Offset', 0.4);
```

Communicate with the instrument. For example, output signals. In this example, the `enableOutput` function enables the function generator to produce a signal that appears at the output connector.

```
% Enable the output of signals.
enableOutput(myFGen);
```

When you are done, disable the output.

```
% Disable the output of signals.
disableOutput(myFGen);
```

Close the session and remove it from the workspace.

```
disconnect(myFGen);
delete myFGen;
clear myFGen;
```

These examples used a standard waveform type. For examples using an arbitrary waveform, see “Generate Standard Waveforms Using the Quick-Control Function Generator” on page 14-31.

See Also

Topics

“Quick-Control Function Generator Requirements” on page 14-30

Introduced in R2012a

fgetl

(To be removed) Read line of text from instrument and discard terminator

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
tline = fgetl(obj)
[tline,count] = fgetl(obj)
[tline,count,msg] = fgetl(obj)
[tline,count,msg,datagramaddress,datagramport] = fgetl(obj)
```

Arguments

`obj`	An interface object.
`tline`	The text read from the instrument, excluding the terminator.
`count`	The number of values read, including the terminator.
`msg`	A message indicating if the read operation was unsuccessful.
`datagramaddress`	The datagram address.
`datagramport`	The datagram port.

Description

`tline = fgetl(obj)` reads one line of text from the instrument connected to `obj`, and returns the data to `tline`. The returned data does not include the terminator with the text line. To include the terminator, use `fgets`.

`[tline,count] = fgetl(obj)` returns the number of values read to `count`.

`[tline,count,msg] = fgetl(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

`[tline,count,msg,datagramaddress,datagramport] = fgetl(obj)` returns the remote address and port from which the datagram originated. These values are returned only if `obj` is a UDP object.

Examples

Create the GPIB object `g`, connect `g` to a Tektronix TDS 210 oscilloscope, configure `g` to complete read operations when the End-Of-String character is read, and write the `*IDN?` command with the `fprintf` function. `*IDN?` instructs the scope to return identification information.

```
g = gpib('ni',0,1);
fopen(g)
```

```
g.EOSMode = 'read';
fprintf(g,'*IDN?')
```

Asynchronously read the identification information from the instrument.

```
readasync(g)
g.BytesAvailable
ans =
    56
```

Use `fgetl` to transfer the data from the input buffer to the MATLAB workspace, and discard the terminator.

```
idn = fgetl(g)
idn =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04

length(idn)
ans =
    55
```

Disconnect `g` from the scope, and remove `g` from memory and the workspace.

```
fclose(g)
delete(g)
clear g
```

Tips

Before you can read text from the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read — including the terminator — each time `fgetl` is issued.

Note You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Rules for Completing a Read Operation with fgetl

A read operation with `fgetl` blocks access to the MATLAB Command Window until

- The terminator is read. For serial port, TCPIP, UDP, and VISA-serial objects, the terminator is given by the `Terminator` property. Note that for UDP objects, `DatagramTerminateMode` must be `off`.

For all other interface objects except VISA-RSIB, the terminator is given by the `EOSCharCode` property.

- The EOI line is asserted (GPIB and VXI instruments only).
- A datagram has been received (UDP objects only if `DatagramTerminateMode` is `on`).
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Note You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

More About the GPIB and VXI Terminator

The `EOSCharCode` property value is recognized only when the `EOSMode` property is configured to `read` or `read&write`. For example, if `EOSMode` is configured to `read` and `EOSCharCode` is configured to `LF`, then one of the ways that the read operation terminates is when the line feed character is received.

If `EOSMode` is `none` or `write`, then there is no terminator defined for read operations. In this case, `fgetl` will complete execution and return control to the command line when another criterion, such as a timeout, is met.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See “Transition Your Code to bluetooth Interface” for more information about using the recommended functionality.

`tcpip` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See “Transition Your Code to tcpclient Interface” on page 7-36 or “Transition Your Code to tcpserver Interface” on page 7-42 for more information about using the recommended functionality.

`udp` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See "Transition Your Code to udpport Interface" on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See "Transition Your Code to VISA-GPIB Interface" on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fgets` | `fopen` | `instrhelp`

Properties
BytesAvailable | EOSCharCode | EOSMode | InputBufferSize | Status | Terminator | Timeout | ValuesReceived

Introduced before R2006a

fgets

(To be removed) Read line of text from instrument and include terminator

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see "Compatibility Considerations".

Syntax

```
tline = fgets(obj)
[tline,count] = fgets(obj)
[tline,count,msg] = fgets(obj)
[tline,count,msg,datagramaddress,datagramport] = fgets(obj)
```

Arguments

`obj`	An interface object.
`tline`	The text read from the instrument, including the terminator.
`count`	The number of values read.
`msg`	A message indicating that the read operation did not complete successfully.
`datagramaddress`	The datagram address.
`datagramport`	The datagram port.

Description

`tline = fgets(obj)` reads one line of text from the instrument connected to `obj`, and returns the data to `tline`. The returned data includes the terminator with the text line. To exclude the terminator, use `fgetl`.

`[tline,count] = fgets(obj)` returns the number of values read to `count`.

`[tline,count,msg] = fgets(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

`[tline,count,msg,datagramaddress,datagramport] = fgets(obj)` returns the remote address and port from which the datagram originated. These values are returned only if `obj` is a UDP object.

Examples

Create the GPIB object `g`, connect `g` to a Tektronix TDS 210 oscilloscope, configure `g` to complete read operations when the End-Of-String character is read, and write the `*IDN?` command with the `fprintf` function. `*IDN?` instructs the scope to return identification information.

```
g = gpib('ni',0,1);
fopen(g)
```

```
g.EOSMode = 'read';
fprintf(g,'*IDN?')
```

Asynchronously read the identification information from the instrument.

```
readasync(g)
g.BytesAvailable
ans =
    56
```

Use `fgets` to transfer the data from the input buffer to the MATLAB workspace, and include the terminator.

```
idn = fgets(g)
idn =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
length(idn)
ans =
    56
```

Disconnect `g` from the scope, and remove `g` from memory and the workspace.

```
fclose(g)
delete(g)
clear g
```

Tips

Before you can read text from the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read — including the terminator — each time `fgets` is issued.

Note You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Rules for Completing a Read Operation with fgets

A read operation with `fgets` blocks access to the MATLAB command line until

- The terminator is read. For serial port, TCPIP, UDP, and VISA-serial objects, the terminator is given by the `Terminator` property. Note that for UDP objects, `DatagramTerminateMode` must be `off`.

For all other interface objects except VISA-RSIB, the terminator is given by the `EOSCharCode` property.

- The EOI line is asserted (GPIB and VXI instruments only).
- A datagram has been received (UDP objects only if `DatagramTerminateMode` is `on`).
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Note You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

More About the GPIB and VXI Terminator

The `EOSCharCode` property value is recognized only when the `EOSMode` property is configured to `read` or `read&write`. For example, if `EOSMode` is configured to `read` and `EOSCharCode` is configured to `LF`, then one of the ways that the read operation terminates is when the line feed character is received.

If `EOSMode` is `none` or `write`, then there is no terminator defined for read operations. In this case, `fgets` will complete execution and return control to the command line when another criterion, such as a timeout, is met.

Compatibility Considerations

`serial` object interface will be removed
Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See “Transition Your Code to udpport Interface” on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fgetl` | `fopen` | `instrhelp` | `query`

Properties
BytesAvailable | EOSCharCode | EOSMode | InputBufferSize | Status | Terminator | Timeout | ValuesReceived

Introduced before R2006a

flush

Clear serial port device buffers

Syntax

```
flush(device)
flush(device,"input")
flush(device,"output")
```

Description

`flush(device)` flushes all data from both the input and output buffers of the specified serial port.

`flush(device,"input")` flushes only the input buffer.

`flush(device,"output")` flushes only the output buffer.

Examples

Flush Serial Port Device Inputs and Outputs

Create a connection to a serial port device.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Write some data to the device and view the number of bytes available to be read in the input buffer.

```
write(device,1:5,"uint8")
device.NumBytesAvailable
```

```
ans =

     5
```

Flush both the input and output buffers.

```
flush(device);
```

View the number of bytes available to be read.

```
device.NumBytesAvailable
```

```
ans =

     0
```

The input buffer has no data.

Input Arguments

`device` — Serial port connection
`serialport` object

Serial port connection, specified as a `serialport` object.

Example: `flush(device)` flushes data from the serial port connection `device`.

See Also

Functions
`getpinstatus` | `read` | `serialport` | `write`

Introduced in R2019b

flush

Clear buffers for communication with remote host over TCP/IP

Syntax

```
flush(t)
flush(t,"input")
flush(t,"output")
```

Description

`flush(t)` flushes all data from both the input and output buffers of the remote host specified by the TCP/IP client `t`.

`flush(t,"input")` flushes only the input buffer.

`flush(t,"output")` flushes only the output buffer.

Examples

Flush Remote Host Inputs and Outputs

Create a TCP/IP client called `t`, using the IP address `172.28.154.231` and port `4012`.

```
t = tcpclient("172.28.154.231",4012)
```

```
t = 

  tcpclient with properties:

              Address: '172.28.154.231'
                 Port: 4012
    NumBytesAvailable: 0

  Show all properties, functions
```

Write some data to the remote host and view the number of bytes available to be read in the input buffer.

```
write(t,1:5,"uint8")
t.NumBytesAvailable
```

```
ans = 

     5
```

Flush both the input and output buffers.

```
flush(t)
```

View the number of bytes available to be read.

```
t.NumBytesAvailable
```

```
ans =

     0
```

The input buffer has no data.

Input Arguments

t — TCP/IP client
`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `flush(t)` flushes data from the TCP/IP client `t`.

See Also

Functions
`tcpclient` | `read` | `write`

Introduced in R2020b

flush

Clear buffers for communication using TCP/IP server

Syntax

```
flush(t)
flush(t,"input")
flush(t,"output")
```

Description

`flush(t)` flushes all data from both the input and output buffers of the client connected to the TCP/IP server `t`.

`flush(t,"input")` flushes only the input buffer.

`flush(t,"output")` flushes only the output buffer.

Examples

Flush TCP/IP Server Inputs and Outputs

Create a TCP/IP server on port 4000.

```
server = tcpserver(4000)
```

```
server =
  TCPServer with properties:

        ServerAddress: "::"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same port number you use to create `server`.

```
client = tcpclient("localhost",4000);
```

Write some data to the client and view the number of bytes available to be read in the server input buffer.

```
write(client,1:5,"uint8")
server.NumBytesAvailable
```

```
ans = 5
```

Flush both the input and output buffers of the server.

```
flush(server)
```

View the number of bytes available to be read.

```
server.NumBytesAvailable
```

```
ans = 0
```

The input buffer has no data.

Input Arguments

t — TCP/IP server
`tcpserver` object

TCP/IP server, specified as a `tcpserver` object.

Example: `flush(t)` flushes data from the TCP/IP server `t`.

See Also

`tcpserver` | `read` | `write`

Introduced in R2021a

flush

Clear UDP socket buffers

Syntax

```
flush(u)
flush(u,"input")
flush(u,"output")
```

Description

`flush(u)` flushes all data from both the input and output buffers of the specified UDP socket.

`flush(u,"input")` flushes only the input buffer.

`flush(u,"output")` flushes only the output buffer.

Examples

Clear UDP Buffer Data

Create a `udpport` object, and clear its buffer.

Clear only the input buffer.

```
u = udpport;
% :
flush(u,"input")
```

Clear both the input and output buffers.

```
flush(u)
```

Input Arguments

u — UDP socket
`udpport` object

UDP socket, specified as a `udpport` object.

Example: `u = udpport`

Data Types: `udpport object`

See Also

Functions
`udpport`

Introduced in R2020b

flush

Clear buffers for communication with VISA resource

Syntax

```
flush(v)
flush(v,"input")
flush(v,"output")
```

Description

`flush(v)` flushes all data from both the input and output buffers of the VISA resource `v` and clears the hardware output buffer of the instrument.

`flush(v,"input")` flushes only the input buffer of the VISA resource.

`flush(v,"output")` flushes only the output buffer of the VISA resource.

Examples

Flush VISA Resource Inputs and Outputs

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Write some data to the device and view the number of bytes available to be read in the input buffer.

```
write(v,1:5,"uint8")
v.NumBytesAvailable
```

```
ans =

     5
```

Flush both the input and output buffers.

```
flush(v)
```

View the number of bytes available to be read.

```
v.NumBytesAvailable
```

```
ans =

     0
```

The input buffer has no data.

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `flush(v)` flushes data from the VISA resource `v`.

See Also

`visadev` | `read` | `write`

Introduced in R2021a

flushinput

(To be removed) Remove data from input buffer

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see "Compatibility Considerations".

Syntax

```
flushinput(obj)
```

Arguments

`obj`	An interface object or an array of interface objects.

Description

`flushinput(obj)` removes data from the input buffer associated with `obj`.

Tips

After the input buffer is flushed, the `BytesAvailable` property is automatically configured to `0`.

If `flushinput` is called during an asynchronous (nonblocking) read operation, the data currently stored in the input buffer is flushed and the read operation continues. You can read data asynchronously from the instrument using the `readasync` function.

The input buffer is automatically flushed when you connect an object to the instrument with the `fopen` function.

You can clear the output buffer with the `flushoutput` function. You can clear the hardware buffer for GPIB and VXI instruments with the `clrdevice` function.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See "Transition Your Code to udpport Interface" on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

gpib object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See "Transition Your Code to VISA-GPIB Interface" on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`clrdevice` | `flushoutput` | `fopen` | `readasync`

Properties
BytesAvailable

Introduced before R2006a

flushoutput

(To be removed) Remove data from output buffer

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see "Compatibility Considerations".

Syntax

```
flushoutput(obj)
```

Arguments

`obj`	An interface object or an array of interface objects.

Description

`flushoutput(obj)` removes data from the output buffer associated with `obj`.

Tips

After the output buffer is flushed, the `BytesToOutput` property is automatically configured to `0`.

If `flushoutput` is called during an asynchronous (nonblocking) write operation, the data currently stored in the output buffer is flushed and the write operation is aborted. Additionally, the callback function specified for the `OutputEmptyFcn` property is executed. You can write data asynchronously to the instrument using the `fprintf` or `fwrite` functions.

The output buffer is automatically flushed when you connect an object to the instrument with the `fopen` function.

You can clear the input buffer with the `flushinput` function. You can clear the hardware buffer for GPIB and VXI instruments with the `clrdevice` function.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See "Transition Your Code to udpport Interface" on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

gpib object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See "Transition Your Code to VISA-GPIB Interface" on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`clrdevice` | `flushinput` | `fopen` | `fprintf` | `fwrite`

Properties
BytesToOutput | OutputEmptyFcn

Introduced before R2006a

fopen

Connect interface object to instrument

Syntax

```
fopen(obj)
```

Arguments

`obj`	An interface object or an array of interface objects.

Description

`fopen(obj)` connects `obj` to the instrument.

Examples

This example creates the GPIB object `g`, connects `g` to the instrument using `fopen`, writes and reads text data, and then disconnects `g` from the instrument.

```
g = gpib('ni',0,1);
fopen(g)
fprintf(g,'*IDN?')
idn = fscanf(g);
fclose(g)
```

Tips

Before you can perform a read or write operation, `obj` must be connected to the instrument with the `fopen` function. When `obj` is connected to the instrument

- Data remaining in the input buffer or the output buffer is flushed.
- The `Status` property is set to `open`.
- The `BytesAvailable`, `ValuesReceived`, `ValuesSent`, and `BytesToOutput` properties are set to 0.

An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the instrument. You can connect only one interface object to a given instrument. For example, on a Windows machine you can connect only one serial port object to an instrument associated with the COM1 port. Similarly, you can connect only one GPIB object to an instrument with a given board index, primary address, and secondary address.

Some properties are read-only while the interface object is connected, and must be configured before using `fopen`. Examples include `InputBufferSize` and `OutputBufferSize`. Refer to the property reference pages or use the `propinfo` function to determine which properties have this constraint.

The values for some properties are verified only after `obj` is connected to the instrument. If any of these properties are incorrectly configured, an error is returned when `fopen` is issued and `obj` is not

connected to the instrument. Properties of this type include `BaudRate` and `SecondaryAddress`, and are associated with instrument settings.

Compatibility Considerations

`serial` object interface will be removed
Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See "Transition Your Code to udpport Interface" on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

gpib object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fclose` | `instrhelp` | `propinfo`

Properties
BytesAvailable | BytesToOutput | Status | ValuesReceived | ValuesSent

Introduced before R2006a

fprintf

(To be removed) Write text to instrument

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see "Compatibility Considerations".

Syntax

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

Arguments

`obj`	An interface object.
`'cmd'`	The string written to the instrument.
`'`*`format`*`'`	C language conversion specification.
`'`*`mode`*`'`	Specifies whether data is written synchronously or asynchronously.

Description

`fprintf(obj,'cmd')` writes the string `cmd` to the instrument connected to `obj`. The default format is `%s\n`. The write operation is synchronous and blocks the command line until execution is complete.

`fprintf(obj,'`*`format`*`','cmd')` writes the string using the format specified by *`format`*.

`format` is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sprintf` file I/O format specifications or a C manual for more information.

`fprintf(obj,'cmd','`*`mode`*`')` writes the string with command-line access specified by *`mode`*. If *`mode`* is `sync`, `cmd` is written synchronously and the command line is blocked. If *`mode`* is `async`, `cmd` is written asynchronously and the command line is not blocked. If *`mode`* is not specified, the write operation is synchronous.

`fprintf(obj,'`*`format`*`','cmd','`*`mode`*`')` writes the string using the specified format. If *`mode`* is `sync`, `cmd` is written synchronously. If *`mode`* is `async`, `cmd` is written asynchronously.

Examples

Create the serial port object `s`, connect `s` on a Windows machine to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
settings = fscanf(s)
settings =
9600;1;0;NONE;LF
```

Because the default format for `fprintf` is `%s\n`, the terminator specified by the `Terminator` property was automatically written. However, in some cases you might want to suppress writing the terminator. To do so, you must explicitly specify a format for the data that does not include the terminator, or configure the terminator to empty.

```
fprintf(s,'%s','RS232?')
```

Tips

Before you can write text to the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the instrument.

The `ValuesSent` property value is increased by the number of values written each time `fprintf` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

`fprintf` function will return an error message if you set the `flowcontrol` property to `hardware` on a serial object, and a hardware connection is not detected. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Note If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is `off`, there is a problem on the remote device side. If `ClearToSend` is `on`, there is a hardware `FlowControl` device prepared to receive data and you can execute `fprintf`.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Synchronous Versus Asynchronous Write Operations

By default, text is written to the instrument synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes,

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Rules for Completing a Write Operation with fprintf

A write operation using `fprintf` completes when

- The specified data is written.
- The time specified by the `Timeout` property passes.

Rules for Writing the Terminator

For serial port, TCPIP, UDP, and VISA-serial objects, all occurrences of `\n` in `cmd` are replaced with the `Terminator` property value. Therefore, when using the default format `%s\n`, all commands written to the instrument will end with this property value.

For GPIB, VISA-GPIB, VISA-VXI, and VISA-GPIB-VXI objects, all occurrences of `\n` in `cmd` are replaced with the `EOSCharCode` property value if the `EOSMode` property is set to `write` or `read&write`. For example, if `EOSMode` is set to `write` and `EOSCharCode` is set to `LF`, then all occurrences of `\n` are replaced with a line feed character. Additionally, for GPIB objects, the End Or Identify (EOI) line is asserted when the terminator is written out.

Note The terminator required by your instrument will be described in its documentation.

Compatibility Considerations

`serial` object interface will be removed
Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See "Transition Your Code to udpport Interface" on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See "Transition Your Code to VISA-GPIB Interface" on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fopen` | `fwrite` | `instrhelp` | `query` | `sprintf`

Properties
BytesToOutput | EOSCharCode | EOSMode | OutputBufferSize | OutputEmptyFcn | Status | TransferStatus | ValuesSent

Introduced before R2006a

fread

Read binary data from instrument

Syntax

```
A = fread(obj)
A = fread(obj,size)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
[A,count,msg,datagramaddress] = fread(obj,...)
[A,count,msg,datagramaddress,datagramport] = fread(obj,...)
```

Arguments

`obj`	An interface object.
`size`	The number of values to read.
`'precision'`	The number of bits read for each value, and the interpretation of the bits as character, integer, or floating-point values.
`A`	Binary data returned from the instrument.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.
`datagramaddress`	The address of the datagram sender.
`datagramport`	The port of the datagram sender.

Description

`A = fread(obj)` and `A = fread(obj,size)` read binary data from the instrument connected to `obj`, and returns the data to `A`. The maximum number of values to read is specified by `size`. If `size` is not specified, the maximum number of values to read is determined by the object's `InputBufferSize` property. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m–by–n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. A value is defined as a byte multiplied by the *`precision`* (see below).

If `obj` is a UDP object and `DatagramTerminateMode` is `off`, the `size` value is honored. If `size` is less than the length of the datagram, only `size` values are read. If `size` is greater than the length of the datagram, a warning is issued stating that a complete datagram was read before `size` values was reached.

`A = fread(obj,size,'precision')` reads binary data with precision specified by *`precision`*.

`precision` controls the number of bits read for each value and the interpretation of those bits as integer, floating-point, or character values. If `precision` is not specified, `uchar` (an 8-bit unsigned character) is used. By default, numeric values are returned in double-precision arrays. The supported values for `precision` are listed below in Tips on page 24-160.

`[A,count] = fread(...)` returns the number of values read to `count`.

`[A,count,msg] = fread(...)` returns a warning message to `msg` if the read operation was unsuccessful.

`[A,count,msg,datagramaddress] = fread(obj,...)` returns the datagram address to `datagramaddress` if `obj` is a UDP object. If more than one datagram is read, `datagramaddress` is `''`.

`[A,count,msg,datagramaddress,datagramport] = fread(obj,...)` returns the datagram port to `datagramport` if `obj` is a UDP object. If more than one datagram is read, `datagramport` is [ ].

Tips

Before you can read data from the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read, each time `fread` is issued.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Rules for Completing a Binary Read Operation

A read operation with `fread` blocks access to the MATLAB Command Window until

- The specified number of values is read. For UDP objects, `DatagramTerminateMode` must be `off`.
- The time specified by the `Timeout` property passes.
- A datagram is received (for UDP objects only when `DatagramTerminateMode` is `on`).
- The input buffer is filled.
- The EOI line is asserted (GPIB and VXI instruments only).
- The `EOSCharCode` is received (GPIB and VXI instruments only).

More About the GPIB and VXI Terminator

The `EOSCharCode` property value is recognized only when the `EOSMode` property is configured to `read` or `read&write`. For example, if `EOSMode` is configured to `read` and `EOSCharCode` is

configured to `LF`, then one of the ways that the read operation terminates is when the line feed character is received.

If `EOSMode` is `none` or `write`, then there is no terminator defined for read operations. In this case, `fread` will complete execution and return control to the command when another criterion, such as a timeout, is met.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

Compatibility Considerations

`serial` object interface will be removed
Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See “Transition Your Code to bluetooth Interface” for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See “Transition Your Code to tcpclient Interface” on page 7-36 or “Transition Your Code to tcpserver Interface” on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See “Transition Your Code to udpport Interface” on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

gpib object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fgetl` | `fgets` | `fopen` | `fscanf` | `instrhelp`

Properties
BytesAvailable | InputBufferSize | Status | ValuesReceived

Introduced before R2006a

fscanf

(To be removed) Read data from instrument, and format as text

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
A = fscanf(obj)
A = fscanf(obj,'format')
A = fscanf(obj,'format',size)
[A,count] = fscanf(...)
[A,count,msg] = fscanf(...)
[A,count,msg,datagramaddress] = fscanf(obj,...)
[A,count,msg,datagramaddress,datagramport] = fscanf(obj,...)
```

Arguments

`obj`	An interface object.
`'format'`	C language conversion specification.
`size`	The number of values to read.
`A`	Data read from the instrument and formatted as text.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.
`datagramaddress`	The address of the datagram sender.
`datagramport`	The port of the datagram sender.

Description

`A = fscanf(obj)` reads data from the instrument connected to `obj`, and returns it to `A`. The data is converted to text using the `%c` format.

`A = fscanf(obj,'format')` reads data and converts it according to *`format`*.

`format` is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sscanf` file I/O format specifications or a C manual for more information.

`A = fscanf(obj,'format',size)` reads the number of values specified by `size`. Valid options for `size` are

`n`	Read at most `n` values into a column vector.
`[m,n]`	Read at most m–by–n values filling an m–by–n matrix in column order.

size cannot be inf, and an error is returned if the specified number of values cannot be stored in the input buffer. If size is not of the form [m,n], and a character conversion is specified, then A is returned as a row vector. You specify the size, in bytes, of the input buffer with the InputBufferSize property. An ASCII value is one byte.

If obj is a UDP object and DatagramTerminateMode is off, the size value is honored. If size is less than the length of the datagram, only size values are read. If size is greater than the length of the datagram, a warning is issued stating that a complete datagram was read before size values was reached.

[A,count] = fscanf(...) returns the number of values read to count.

[A,count,msg] = fscanf(...) returns a warning message to msg if the read operation did not complete successfully.

[A,count,msg,datagramaddress] = fscanf(obj,...) returns the datagram address to datagramaddress if obj is a UDP object. If more than one datagram is read, datagramaddress is ''.

[A,count,msg,datagramaddress,datagramport] = fscanf(obj,...) returns the datagram port to datagramport if obj is a UDP object. If more than one datagram is read, datagramport is [ ].

Examples

Create the serial port object s on a Windows machine and connect s to a Tektronix TDS 210 oscilloscope, which is displaying a sine wave.

```
s = serial('COM1');
fopen(s)
```

Use the fprintf function to configure the scope to measure the peak-to-peak voltage of the sine wave, return the measurement type, and return the peak-to-peak voltage.

```
fprintf(s,'MEASUREMENT:IMMED:TYPE PK2PK')
fprintf(s,'MEASUREMENT:IMMED:TYPE?')
fprintf(s,'MEASUREMENT:IMMED:VAL?')
```

Because the default value for the ReadAsyncMode property is continuous, data associated with the two query commands is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    13
```

Use fscanf to read the measurement type. The operation will complete when the first terminator is read.

```
meas = fscanf(s)
meas =
PK2PK
```

Use fscanf to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```
pk2pk = fscanf(s,'%e',6)
pk2pk =
    2.0200
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

```
fclose(s)
delete(s)
clear s
```

Tips

Before you can read data from the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read — including the terminator — each time `fscanf` is issued.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Rules for Completing a Read Operation with fscanf

A read operation with `fscanf` blocks access to the MATLAB command line until

- The terminator is read. For serial port, TCPIP, UDP, and VISA-serial objects, the terminator is given by the `Terminator` property. If `Terminator` is empty, `fscanf` will complete execution and return control when another criterion is met. For UDP objects, `DatagramTerminateMode` must be `off`.

  For all other interface objects, the terminator is given by the `EOSCharCode` property.
- The time specified by the `Timeout` property passes.
- The number of values specified by `size` is read. For UDP objects, `DatagramTerminateMode` must be `off`.
- A datagram is received (for UDP objects only when `DatagramTerminateMode` is `on`).
- The input buffer is filled.
- The EOI line is asserted (GPIB and VXI instruments only).

More About the GPIB and VXI Terminator

The `EOSCharCode` property value is recognized only when the `EOSMode` property is configured to `read` or `read&write`. For example, if `EOSMode` is configured to `read` and `EOSCharCode` is configured to `LF`, then one of the ways that the read operation terminates is when the line feed character is received.

If `EOSMode` is `none` or `write`, then there is no terminator defined for read operations. In this case, `fscanf` will complete execution and return control to the command when another criterion, such as a timeout, is met.

Compatibility Considerations

`serial` object interface will be removed
Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

`udp` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See "Transition Your Code to udpport Interface" on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fgetl` | `fgets` | `fopen` | `fread` | `instrhelp` | `scanstr` | `sscanf`

Properties
BytesAvailable | BytesAvailableFcn | EOSCharCode | EOSMode | InputBufferSize | Status | Terminator | Timeout | TransferStatus

Introduced before R2006a

fwrite

Write binary data to instrument

Syntax

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

Arguments

`obj`	An interface object.
`A`	The binary data written to the instrument.
'*precision*'	The number of bits written for each value, and the interpretation of the bits as character, integer, or floating-point values.
'*mode*'	Specifies whether data is written synchronously or asynchronously.

Description

`fwrite(obj,A)` writes the binary data `A` to the instrument connected to `obj`.

`fwrite(obj,A,'precision')` writes binary data with precision specified by *precision*.

precision controls the number of bits written for each value and the interpretation of those bits as integer, floating-point, or character values. If *precision* is not specified, `uchar` (an 8-bit unsigned character) is used. The support values for *precision* are listed in “Supported Precisions” on page 24-161.

`fwrite(obj,A,'mode')` writes binary data with command line access specified by *mode*. If *mode* is `sync`, `A` is written synchronously and the command line is blocked. If *mode* is `async`, `A` is written asynchronously and the command line is not blocked. If *mode* is not specified, the write operation is synchronous.

`fwrite(obj,A,'precision','mode')` writes binary data with precision specified by *precision* and command-line access specified by *mode*.

Tips

Before you can write data to the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the instrument.

The `ValuesSent` property value is increased by the number of values written each time `fwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

`fwrite` will return an error message if you set the `FlowControl` property to `hardware` on a serial object, and a hardware connection is not detected. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Note If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is `off`, there is a problem on the remote device side. If `ClearToSend` is `on`, there is a hardware `FlowControl` device prepared to receive data and you can execute `fwrite`.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Synchronous Versus Asynchronous Write Operations

By default, data is written to the instrument synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *`mode`* input argument to be `async`. For asynchronous writes,

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Rules for Completing a Write Operation with fwrite

A binary write operation using `fwrite` completes when

- The specified data is written.
- The time specified by the `Timeout` property passes.

Note The `Terminator` and `EOSCharCode` properties are not used with binary write operations.

Supported Precisions

The supported values for *`precision`* are listed below.

Data Type	Precision	Interpretation
Character	`uchar`	8-bit unsigned character
	`schar`	8-bit signed character
	`char`	8-bit signed or unsigned character

Data Type	Precision	Interpretation
Integer	`int8`	8-bit integer
	`int16`	16-bit integer
	`int32`	32-bit integer
	`uint8`	8-bit unsigned integer
	`uint16`	16-bit unsigned integer
	`uint32`	32-bit unsigned integer
	`short`	16-bit integer
	`int`	32-bit integer
	`long`	32- or 64-bit integer
	`ushort`	16-bit unsigned integer
	`uint`	32-bit unsigned integer
	`ulong`	32- or 64-bit unsigned integer
Floating-point	`single`	32-bit floating point
	`float32`	32-bit floating point
	`float`	32-bit floating point
	`double`	64-bit floating point
	`float64`	64-bit floating point

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

`udp` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See “Transition Your Code to udpport Interface” on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed

Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed

Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions

`fopen` | `fprintf` | `instrhelp`

Properties

OutputBufferSize | OutputEmptyFcn | Status | Timeout | TransferStatus | ValuesSent

Introduced before R2006a

get

Instrument object properties

Syntax

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
```

Arguments

`obj`	An instrument object or an array of instrument objects.
`'PropertyName'`	A property name or a cell array of property names.
`out`	A single property value, a structure of property values, or a cell array of property values.

Description

`get(obj)` returns all property names and their current values to the command line for `obj`. The properties are divided into two sections. The base properties are listed first and the object-specific properties are listed second.

`out = get(obj)` returns the structure `out` where each field name is the name of a property of `obj`, and each field contains the value of that property.

`out = get(obj,'PropertyName')` returns the value `out` of the property specified by *`PropertyName`* for `obj`. If *`PropertyName`* is replaced by a 1-by-n or n-by-1 cell array of character vectors containing property names, then `get` returns a 1-by-n cell array of values to `out`. If `obj` is an array of instrument objects, then `out` will be an m-by-n cell array of property values where m is equal to the length of `obj` and n is equal to the number of properties specified.

Examples

This example illustrates some of the ways you can use `get` to return property values for the GPIB object `g`.

```
g = gpib('ni',0,1);
out1 = get(g);
out2 = get(g,{'PrimaryAddress','EOSCharCode'});
get(g,'EOIMode')
ans =
on
```

Tips

When specifying a property name, you can do so without regard to case, and you can make use of property name completion. For example, if `g` is a GPIB object, then these commands are all valid.

```
out = get(g,'EOSMode');
out = get(g,'eosmode');
out = get(g,'EOSM');
```

See Also

instrhelp | propinfo | set

Introduced before R2006a

geterror

Check and return error message from instrument

Syntax

```
msg = geterror(obj)
```

Arguments

`obj`	A device object.
`msg`	The error message returned from the instrument.

Description

`msg = geterror(obj)` checks the instrument associated with the device object specified by `obj` for an error message. If an error message exists, it is returned to `msg`. The interpretation of `msg` will vary based on the instrument.

Introduced before R2006a

getpinstatus

Get serial pin status

Syntax

`status = getpinstatus(device)`

Description

`status = getpinstatus(device)` gets the serial pin status and returns it as a structure to `status`.

Examples

Get Serial Pin Status

Get the serial pin status for the specified port.

```
device = serialport("COM3",9600);
 ⋮
status = getpinstatus(device)
```

```
status = 

  struct with fields:

      ClearToSend: 1
     DataSetReady: 1
    CarrierDetect: 1
    RingIndicator: 0
```

Get Serial Pin Status Using VISA

Get the serial pin status for the specified port using the VISA-Serial interface.

```
device = visadev("COM3");
 ⋮
status = getpinstatus(device)
```

```
status = 

  struct with fields:

      ClearToSend: 1
     DataSetReady: 1
```

```
    CarrierDetect: 1
    RingIndicator: 0
```

Input Arguments

device — Serial port connection

serialport object | visadev object

Serial port, specified as a serialport object or visadev object.

Example: getpinstatus(device) returns the serial pin status for the serial port connection device.

Example: getpinstatus(device) returns the serial pin status for the VISA-Serial resource device.

Output Arguments

status — Pin status

structure

Pin status, returned as a structure with the logical type fields named ClearToSend, DataSetReady, CarrierDetect, and RingIndicator.

See Also

Functions
serialport | visadev | flush | flush

Introduced in R2019b

getWaveform

Returns waveform displayed on scope

Syntax

```
w = getWaveform(myScope);
w = getWaveform(myScope, 'acquisition', true);
w = getWaveform(myScope, 'acquisition', false);
```

Description

`w = getWaveform(myScope);` returns waveform(s) displayed on the scope screen. Retrieves the waveform(s) from enabled channel(s). By default it downloads the captured waveform from the scope without acquisition.

`w = getWaveform(myScope, 'acquisition', true);` initiates an acquisition and returns waveform(s) from the oscilloscope.

`w = getWaveform(myScope, 'acquisition', false);` gets waveform from the enabled channel without acquisition

This function can only be used with the `oscilloscope` object. You can use the `getWaveform` function to download the current waveform from the scope or to initiate the waveform and capture it. See the examples below for the three possible use cases.

Note You should now use the readWaveform function. In R2017a the name changed from getWaveform to readWaveform. The getWaveform function will continue to be supported.

Examples

Use this example if you have captured the waveform(s) using the oscilloscope's front panel and want to download it to the Instrument Control Toolbox for further analysis.

```
o = oscilloscope()
set (o, 'Resource', 'instrumentResourceString');
connect(o);
w = getWaveform(o);
```

Replace `'instrumentResourceString'` with the resource string for your instrument.

Use this example to get the waveform from a circuit output (without configuring the trigger) and download it to the Instrument Control Toolbox to check it.

```
o = oscilloscope()
set (o, 'Resource', 'instrumentResourceString');
connect(o);
enableChannel(o,'Channel1');
w = getWaveform(o);
```

Replace `'instrumentResourceString'` with the resource string for your instrument.

Use this example to capture synchronized input/output signals of a filter circuit when a certain trigger condition is met, stop the acquisition, and download the waveforms to the Instrument Control Toolbox.

```
o = oscilloscope()
set (o, 'Resource', 'instrumentResourceString');
connect(o);
set (o, 'TriggerMode','normal');
enableChannel(o, {'Channel1','Channel2'});
[w1, w2] = getWaveform(o, 'acquisition', true);
```

Replace `'instrumentResourceString'` with the resource string for your instrument.

See Also

Topics
"The Quick-Control Interfaces" on page 14-20

Introduced in R2011b

gpib

(To be removed) Create GPIB object

Note `gpib` will be removed in a future release. Use `visadev` instead. For more information, see "Compatibility Considerations".

Syntax

```
obj = gpib('vendor',boardindex,primaryaddress)
obj = gpib('vendor',boardindex,primaryaddress,'PropertyName',PropertyValue)
```

Arguments

`'vendor'`	The vendor name.
`boardindex`	The GPIB board index.
`primaryaddress`	The instrument primary address.
`'PropertyName'`	A GPIB property name.
`'PropertyValue'`	A property value supported by *PropertyName*.
`obj`	The GPIB object.

Description

`obj = gpib('vendor',boardindex,primaryaddress)` creates the GPIB object `obj` associated with the board specified by `boardindex`, and the instrument specified by `primaryaddress`. The GPIB hardware is supplied by *vendor*. Supported vendors are given below.

Vendor	Description
`keysight`	Keysight (formerly Agilent Technologies) hardware
`ics`	ICS Electronics hardware
`mcc`	Measurement Computing hardware
`ni`	National Instruments hardware
`adlink`	ADLINK Technology hardware

`obj = gpib('vendor',boardindex,primaryaddress,'PropertyName',PropertyValue)` creates the GPIB object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and `obj` is not created.

Examples

This example creates the GPIB object `g1` associated with a National Instruments board at index 0 with primary address 1, and then connects `g1` to the instrument.

```
g1 = gpib('ni',0,1);
fopen(g1)
```

The `Type`, `Name`, `BoardIndex`, and `PrimaryAddress` properties are automatically configured.

```
g1.Type
ans =
    gpib

g1.Name
ans =
    GPIB0-1

g1.BoardIndex
ans =
    0

g1.PrimaryAddress
ans =
    1
```

To specify the secondary address during object creation,

```
g2 = gpib('ni',0,1,'SecondaryAddress',96);
```

Tips

At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with GPIB objects.

```
instrhelp gpib
```

When you create a GPIB object, these property value are automatically configured:

- `Type` is given by `gpib`.
- `Name` is given by concatenating `GPIB` with the board index and the primary address specified in the `gpib` function. If the secondary address is specified, then this value is also used in `Name`.
- `BoardIndex` and `PrimaryAddress` are given by the values supplied to the `gpib` function.

Note You do not use the GPIB board primary address in the GPIB object constructor syntax. You use the board index, and the instrument address.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, these commands are all valid:

```
g = gpib('ni',0,1,'SecondaryAddress',96);
g = gpib('ni',0,1,'secondaryaddress',96);
g = gpib('ni',0,1,'SECOND',96);
```

Before you can communicate with the instrument, it must be connected to `obj` with the `fopen` function. A connected GPIB object has a `Status` property value of `open`. An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the instrument.

You cannot connect multiple GPIB objects to the same instrument. A GPIB instrument is uniquely identified by its board index, primary address, and secondary address.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Compatibility Considerations

`gpib` function will be removed

Not recommended starting in R2021b

`gpib` and its object properties will be removed in a future release. Use `visadev` and its properties instead.

This example shows how to connect to a GPIB instrument using the recommended functionality.

Functionality	Use This Instead
`g = gpib('ni',0,1,'SecondaryAddress',0);` `fopen(g)`	`v = visadev('GPIB0::1::0::INSTR');`

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

`visadev` | `fopen` | `instrhelp` | `instrhwinfo` | BoardIndex | Name | PrimaryAddress | SecondaryAddress | Status | Type

Introduced before R2006a

i2c

Create I2C object

Description

I2C, or Inter-Integrated Circuit, is a chip-to-chip protocol supporting two-wire communication. An `i2c` object represents a connection between MATLAB and an I2C adapter board. Supported adapters are the Total Phase Aardvark I2C/SPI Host Adapter and the National Instruments USB-845x adapter board. The adapter has one or more sensor chips connected to it. MATLAB sends commands to the adapter board, which is the I2C controller device, in order to communicate with the chip, which is the I2C peripheral device. The `i2c` object in MATLAB always has the role of I2C controller and cannot be used in the peripheral role. Use `fread` and `fwrite` on the `i2c` object to communicate with the chip.

Creation

Syntax

`i2cobj = i2c(vendor,boardIndex,remoteAddress)`

Description

`i2cobj = i2c(vendor,boardIndex,remoteAddress)` creates an `i2c` object associated with `vendor`, `boardIndex`, and `remoteAddress`.

- `vendor` must be either `'Aardvark'`, for use with the Total Phase Aardvark adapter, or `'NI845x'`, for use with the NI USB-845x adapter board. This input sets the “Vendor” on page 24-0 property.
- `boardIndex` specifies the board index of the adapter board and is `0` if you have only one adapter plugged into your computer. This input sets the “BoardIndex” on page 24-0 property.
- `remoteAddress` specifies the hex number address of the I2C peripheral device with which to communicate and is found in documentation or data sheet of the chip. This input sets the “RemoteAddress” on page 24-0 property.

You can communicate with multiple I2C peripheral devices on the same adapter using a single `i2c` object. To communicate with a different I2C peripheral device, first create the object and use `fopen` to open a connection to the adapter. Then, change the `RemoteAddress` property to the address of the appropriate peripheral device. You can now use `fwrite` and `fread` to communicate with the specified peripheral device. For an example of this workflow, see “Communicate with Multiple Peripheral Devices from NI USB-845x Adapter” on page 24-186.

Properties

`BoardIndex` — Board index
integer

Board index of the adapter board, specified as an integer. If you have only one adapter plugged into your computer, the board index number is `0`. If you have multiple adapters plugged in, each board is

assigned a different board index number. Determine the board index using `instrhwinfo('i2c','Aardvark')` or `instrhwinfo('i2c','NI845x')`. This property can be set only at object creation.

Example: `i2cobj = i2c('Aardvark',1,'50h')` connects to the Aardvark adapter with an index value of 1.

Data Types: `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`BoardSerial` — Unique identifier
character vector

This property is read-only.

Unique identifier of the I2C controller communication device, specified as a character vector.

Example: `i2cobj.BoardSerial` returns the unique identifier of the Total Phase Aardvark adapter or NI USB-845x adapter.

Data Types: `char`

`BitRate` — Bit rate
100 (default) | positive integer

Bit rate of the adapter hardware, specified as a positive integer in kHz. Use a bit rate that is supported by the adapter and chips. The default is 100 kHz for both the Aardvark and USB-845x adapters.

Example: `i2cobj.BitRate = 50` sets the bit rate to 50 kHz.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

`RemoteAddress` — I2C peripheral device address
numeric scalar | hex number | character vector | string scalar

I2C peripheral device address, specified as a number between 0 and 127 (inclusive), hex number, character vector, or string scalar. To identify the address of the chip, consult its documentation or data sheet. You can also find the address by scanning for instruments in the Test & Measurement Tool. In the tool, right-click the **I2C** node and select **Scan for I2C adaptors**. Any chips found by the scan is listed in the hardware tree. The listing includes the remote address of the chip.

You must specify this property during object creation and you can change it after object creation by using dot notation. Specify the peripheral address as a hexadecimal value in one of the following ways.

- Append `h` to the end of the hex number as a character vector or string scalar.
- Use the prefix `0x` at the beginning of the hex number.
- Use `hex2dec` with the hex number as the input argument.

To communicate with multiple I2C peripheral devices from one `i2c` object, set `RemoteAddress` to the appropriate peripheral device address. For an example of this workflow, see “Communicate with Multiple Peripheral Devices from NI USB-845x Adapter” on page 24-186.

Note When reading this property using dot notation, it is returned as a numeric scalar representing the hex number. To convert back to the hex number, use `dec2hex`. For example,

`dec2hex(i2cObj.RemoteAddress)` returns the I2C peripheral address as a hex number for the `i2c` object `i2cObj`.

Example: `i2cobj = i2c('Aardvark',0,'50h')` and `i2cobj.RemoteAddress = '50h'` specify an address of 50 hex.

Example: `i2cobj = i2c('Aardvark',0,0x50)` and `i2cobj.RemoteAddress = 0x50` specify an address of 50 hex.

Example: `i2cobj = i2c('Aardvark',0,hex2dec('50'))` and `i2cobj.RemoteAddress = hex2dec('50')` specify an address of 50 hex.

Data Types: `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `char` | `string`

`Vendor` — I2C adapter vendor
`'Aardvark'` | `'NI845x'`

I2C adapter vendor, specified as `'Aardvark'` or `'NI845x'`. The vendor is `'Aardvark'` if you are using the Total Phase Aardvark adapter and `'NI845x'` if you are using the NI USB-845x adapter. This property can be set only at object creation.

Example: `i2cobj = i2c('Aardvark',0,'50h')` connects to a Total Phase Aardvark adapter.

Example: `i2cobj = i2c('NI845x',0,'50h')` connects to a NI USB-845x adapter.

Data Types: `char` | `string`

`TargetPower` — Power to Aardvark adapter
`'none'` (default) | `'both'`

Power to Aardvark adapter, specified as `'none'` or `'both'`. You can set this property for Aardvark adapters only. The value `'both'` means power to both lines, if supported. The value `'none'` means power to no lines.

Note If Target Power is off in the Total Phase Control Center Serial Software, you might receive an error when you try to connect to the Aardvark adapter using `fopen`. To manually set the option, open the software and select the **Target Power** check box.

Example: `i2cobj.TargetPower = 'both'` provides power to both lines on the Aardvark adapter.

Data Types: `char` | `string`

`PullupResistors` — Enabled pullup resistors
`'both'` (default) | `'none'`

Enabled pullup resistors, specified as `'both'` or `'none'`. The value `'both'` enables 2k pullup resistors to protect hardware in the I2C device, if supported.

Devices differ in their use of pullups. The Aardvark and NI USB-8452 adapters have internal pullup resistors to tie both bus lines to VDD, and can be set programmatically. The NI USB-8451 adapter does not have this type of internal pullup resistor and, therefore, requires external pullups. Consult your device documentation to ensure that you are using the correct pullups.

Example: `i2cobj.PullupResistors = 'none'` disables pullup resistors.

Data Types: `char` | `string`

Object Functions

fopen Connect interface object to instrument
fread Read binary data from instrument
fwrite Write binary data to instrument
fclose Disconnect interface object from instrument
record Record data and event information to file

Examples

Communicate with Total Phase Aardvark Adapter

Communicate with an AT24C02 EEPROM chip on a circuit board, with an address of 50 hex and a board index of 0, using the Aardvark adapter.

Ensure that the Aardvark adapter is installed so that you can use the `i2c` interface, and then view the adapter properties.

```
instrhwinfo('i2c')
instrhwinfo('i2c','Aardvark')
```

```
ans =

  HardwareInfo with properties:

    InstalledAdaptors: {'Aardvark'  'NI845x'}
       JarFileVersion: 'Version 4.1'

Access to your hardware may be provided by a support package. Go to the Support Package Installe



ans =

  HardwareInfo with properties:

             AdaptorDllName: 'C:\Program Files\MATLAB\R2019b\toolbox\instrument\instrumentadapto
          AdaptorDllVersion: 'Version 4.1'
                AdaptorName: 'Aardvark'
              BoardIdsInUse: [1×0 double]
          InstalledBoardIDs: 0
       DetectedBoardSerials: {'2237482577 (BoardIndex: 0)'}
      ObjectConstructorName: 'i2c('Aardvark', BoardIndex, RemoteAddress);'
              VendorDllName: 'aardvark.dll'
    VendorDriverDescription: 'Total Phase I2C Driver'

Access to your hardware may be provided by a support package. Go to the Support Package Installe
```

Create an `i2c` object named `i2cobj` with the `Vendor 'Aardvark'`, `BoardIndex` of `0`, and `RemoteAddress` of `50h`.

```
i2cobj = i2c('Aardvark',0,'50h')
```

```
   I2C Object : I2C-0-50h
```

```
   Communication Settings
      BoardIndex          0
      BoardSerial         0
      BitRate:            100 kHz
      RemoteAddress:      50h
      Vendor:             aardvark

   Communication State
      Status:             closed
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
```

Connect to the chip.

```
fopen(i2cobj)
```

Note If Target Power is off in the Total Phase Control Center Serial Software, you might receive an error when you try to connect to the Aardvark adapter using `fopen`. To manually set the option, open the software and select the **Target Power** check box.

Write `'Hello World!'` to the EEPROM chip. Data is written page-by-page in I2C. Each page contains eight bytes. The page address needs to be mentioned before every byte of data written.

The first byte of the string `'Hello World!'` is `'Hello Wo'`. Its page address is 0.

```
fwrite(i2cobj,[0 'Hello Wo'])
```

The second byte of the string `'Hello World!'` is `'rld!'`. Its page address is 8.

```
fwrite(i2cobj,[8 'rld!'])
```

To start reading from the first byte of the first page, write a zero to the `i2c` object.

```
fwrite(i2cobj,0)
```

Read data back from the chip using the `fread` function. The chip returns the characters sent to it.

```
char(fread(i2cobj,12))'
```

```
ans =

    'Hello World!'
```

Disconnect the I2C device.

```
fclose(i2cobj)
```

Clear the object from the workspace.

```
clear i2cobj
```

Communicate with Multiple Peripheral Devices from NI USB-845x Adapter

Communicate with multiple I2C peripheral devices from the same NI USB-845x adapter. You can read from multiple peripheral devices using the Aardvark adapter as well. In this example, the two peripheral devices are two sensor chips on a circuit board, with addresses of 62 hex and 53 hex. The board index is 0. The NI USB-845x adapter board is plugged into the computer (via the USB port), and a circuit board containing the sensor chips is connected to the host adapter board via wires.

Ensure that the NI USB-845x adapter is installed so that you can use the `i2c` interface, and then view the adapter properties.

```
instrhwinfo('i2c')
instrhwinfo('i2c','NI845x')
```

```
ans = 

  HardwareInfo with properties:

    InstalledAdaptors: {'Aardvark'  'NI845x'}
       JarFileVersion: 'Version 4.1'

Access to your hardware may be provided by a support package. Go to the Support Package Installe



ans = 

  HardwareInfo with properties:

             AdaptorDllName: 'C:\Program Files\MATLAB\R2019b\toolbox\instrument\instrumentadapto
          AdaptorDllVersion: 'Version 4.1'
                AdaptorName: 'NI845x'
              BoardIdsInUse: [1×0 double]
          InstalledBoardIDs: [1×0 double]
       DetectedBoardSerials: {0×1 cell}
      ObjectConstructorName: 'i2c('NI845x', BoardIndex, RemoteAddress);'
              VendorDllName: 'Ni845x.dll'
    VendorDriverDescription: 'National Instruments NI USB 845x Driver'

Access to your hardware may be provided by a support package. Go to the Support Package Installe
```

Create an `i2c` object named `i2cobj` with the `Vendor` `'NI845x'`, `BoardIndex` of `0`, and `RemoteAddress` of `62h`. This address is the remote address of the first I2C peripheral device to which you connect.

```
i2cobj = i2c('NI845x',0,'62h')
```

```
   I2C Object : I2C-0-62h

   Communication Settings
      BoardIndex          0
      BoardSerial         0
      BitRate:            100 kHz
      RemoteAddress:      62h
      Vendor:             NI845x
```

```
   Communication State
      Status:             closed
      RecordStatus:       off

   Read/Write State
      TransferStatus:     idle
```

Open a connection to the NI USB-845x adapter. This connection is to the sensor chip with the `RemoteAddress` specified during object creation.

```
fopen(i2cobj)
```

Write to the sensor chip. You need to read the documentation or data sheet of the chip to find the remote address and other information about the chip. In this case, open the chip registry by sending it a `0`.

```
fwrite(i2cobj,0)
```

Read data back from the chip using the `fread` function. By sending one byte, you can read back the device ID registry. For this chip, the read-only device ID registry is 229.

```
fread(i2cobj,1)
```

```
ans =

   229
```

Switch to the second sensor chip by setting the `RemoteAddress` to `53h`. This address is the remote address of the second I2C peripheral device to which you connect. You do not need to reopen a connection with the adapter.

```
i2cobj.RemoteAddress = '53h';
```

Write to and read from the sensor chip. Because this chip is identical to the first chip, its device ID registry is also 229.

```
fwrite(i2cobj,0)
```

```
fread(i2cobj,1)
```

```
ans =

   229
```

Disconnect the I2C device.

```
fclose(i2cobj)
```

Clear the object from the workspace.

```
clear i2cobj
```

See Also

Topics

"I2C Interface Overview" on page 9-2
"Configuring I2C Communication" on page 9-3

"Transmitting Data Over the I2C Interface" on page 9-6

Introduced in R2012a

icdevice

Create device object

Syntax

```
obj = icdevice('driver', hwobj)
obj = icdevice('driver', 'RsrcName')
obj = icdevice('driver')
obj = icdevice('driver', hwobj, 'P1', V1, 'P2', V2,...)
obj = icdevice('driver', 'RsrcName','P1', V1, 'P2', V2,...)
obj = icdevice('driver','P1', V1, 'P2', V2,...)
```

Arguments

`driver`	A MATLAB instrument driver.
`hwobj`	An interface object.
`RsrcName`	VISA resource name.
`'P1', 'P2',...`	Device-specific property names.
`V1, V2,...`	Property values supported by corresponding `P1, P2,...`.
`obj`	A device object.

Description

`obj = icdevice('driver', hwobj)` creates the device object `obj`. The instrument-specific information is defined in the MATLAB interface instrument driver, `driver`. Communication to the instrument is done through the interface object, `hwobj`. The interface object can be a serial port, GPIB, VISA, TCPIP, or UDP object. If `driver` does not exist or if `hwobj` is invalid, the device object is not created.

Device objects may also be used with VXI*plug&play* and Interchangeable Virtual Instrument (IVI) drivers. To use these drivers, you must first have a MATLAB instrument driver wrapper for the underlying VXI*plug&play* or IVI driver. If the MATLAB instrument driver wrapper does not already exist, it may be created using `makemid` or `midedit`. Note that `makemid` or `midedit` only needs to be used once to create the MATLAB instrument driver wrapper.

`obj = icdevice('driver', 'RsrcName')` creates a device object `obj`, using the MATLAB instrument driver, `driver`. The specified `driver` must be a MATLAB VXI*plug&play* instrument driver or MATLAB IVI instrument driver. Communication to the instrument is done through the resource specified by `rsrcname`. For example, all VXI*plug&play*, and many IVI drivers require VISA resource names for `rsrcname`.

`obj = icdevice('driver')` constructs a device object `obj`, using the MATLAB instrument driver, `driver`. The specified `driver` must be a MATLAB IVI instrument driver, and the underlying IVI driver must be referenced using a logical name.

`obj = icdevice('driver', hwobj, 'P1', V1, 'P2', V2,...)`, `obj = icdevice('driver', 'RsrcName','P1', V1, 'P2', V2,...)`, and `obj =`

`icdevice('driver','P1', V1, 'P2', V2,...)`, construct a device object, `obj`, with the specified property values. If an invalid property name or property value is specified, the object will not be created.

Note that the parameter-value pairs can be in any format supported by the `set` function: parameter-value character vector pairs, structures, and parameter-value cell array pairs.

Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, these commands are all valid and equivalent:

```
d = icdevice('tektronix_tds210',g,'ObjectVisibility','on');
d = icdevice('tektronix_tds210',g,'objectvisibility','on');
d = icdevice('tektronix_tds210',g,'ObjectVis','on');
```

Note About Deploying Code

When using IVI-C or VXI Plug&Play drivers, executing your code will generate additional file(s) in the folder specified by executing the following code at the MATLAB prompt:

```
fullfile(tempdir,'ICTDeploymentFiles',sprintf('R%s',version('-release')))
```

On all supported platforms, a file with the name `MATLABPrototypeFor<driverName>.m` is generated, where `<driverName>` the name of the IVI-C or VXI Plug&Play driver. With 64-bit MATLAB on Windows, a second file by the name `<driverName>_thunk_pcwin64.dll` is generated. When creating your deployed application or shared library, manually include these generated files. If using the `icdevice` function, remember to also manually include the MDD-file in the deployed application or shared library. For more information on including additional files refer to the MATLAB Compiler documentation.

Examples

The first example creates a device object for a Tektronix TDS 210 oscilloscope that is connected to a MCC GPIB board, using a MATLAB interface object and MATLAB interface instrument driver.

```
g = gpib('mcc',0,2);
d = icdevice('tektronix_tds210',g);
```

Connect to the instrument.

```
connect(d);
```

List the oscilloscope settings that can be configured.

```
props = set(d);
```

Get the current configuration of the oscilloscope.

```
values = get(d);
```

Disconnect from the instrument and clean up.

```
disconnect(d);
delete([d g]);
```

The second example creates a device object for a Tektronix TDS 210 oscilloscope using a MATLAB VXI*plug&play* instrument driver.

This example assumes that the `'tktds5k'` VXI*plug&play* driver is installed on your system.

This first step is necessary only if a MATLAB VXI*plug&play* instrument driver for the tktds5k does not exist on your system.

```
makemid('tktds5k', 'Tktds5kMATLABDriver');
```

Construct a device object that uses the VXI*plug&play* driver. The instrument is assumed to be located at GPIB primary address 2.

```
d = icdevice('Tktds5kMATLABDriver', 'GPIB0::2::INSTR');
```

Connect to the instrument.

```
connect(d);
```

List the oscilloscope settings that can be configured.

```
props = set(d);
```

Get the current configuration of the oscilloscope.

```
values = get(d);
```

Disconnect from the instrument and clean up.

```
disconnect(d);
delete(d);
```

Tips

At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with device objects.

```
instrhelp icdevice
```

When you create a device object, these property values are automatically configured:

- `Interface` specifies the interface used to communicate with the instrument. For device objects created using interface objects, it is that interface object. For VXI*plug&play* and IVI-C, this is the session handle to the driver session. For MATLAB instrument drivers, this is the handle to the driver's default COM interface.
- `LogicalName` is an IVI logical name. For non-IVI drivers, it is empty.
- `Name` is given by concatenating the instrument type with the name of the instrument driver.
- `RsrcName` is the full VISA resource name for VXI*plug&play* and IVI drivers. For MATLAB interface drivers, `RsrcName` is an empty character vector.
- `Type` is the instrument type, if known (for example, `scope` or `multimeter`).

To communicate with the instrument, the device object must be connected to the instrument with the `connect` function. When the device object is constructed, the object's `Status` property is `closed`. Once the device object is connected to the instrument with the `connect` function, the `Status` property is configured to `open`.

Note ICDEVICE is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

See Also

`connect` | `disconnect` | `instrhelp` | Status

Introduced before R2006a

inspect

Open Property Inspector

Syntax

```
inspect(obj)
```

Arguments

`obj`	An instrument object or an array of instrument objects.

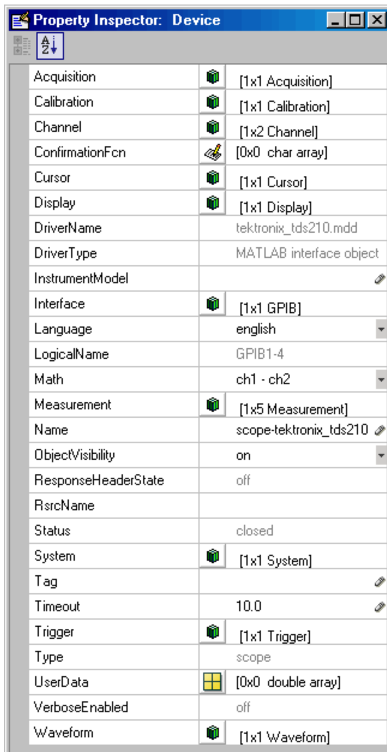
Description

`inspect(obj)` opens the Property Inspector and allows you to inspect and set properties for instrument object `obj`.

Tips

You can also open the Property Inspector via the Workspace browser by right-clicking an instrument object and selecting **Call Property Inspector** from the context menu, or by double-clicking the object.

Below is a Property Inspector for a device object that communicates with a Tektronix TDS 210 oscilloscope.



Introduced before R2006a

instrcallback

Display event information when event occurs

Syntax

`instrcallback(obj, event)`

Arguments

`obj`	An instrument object.
`event`	The event that caused the callback to execute.

Description

`instrcallback(obj, event)` displays a message that contains the event type, the time the event occurred, and the name of the instrument object that caused the event to occur.

For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed. For trigger events, the trigger line is also displayed. For datagram received events, the number of bytes received and the datagram address and port are also displayed.

Note Using this callback for numbers greater than 127 with a terminator is not supported.

Examples

The following example creates the serial port objects `s` on a Windows machine, and configures `s` to execute `instrcallback` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

```
s = serial('COM1');
set(s,'OutputEmptyFcn',@instrcallback)
fopen(s)
fprintf(s,'*IDN?','async')
```

The resulting display from `instrcallback` is shown below.

```
OutputEmpty event occurred at 08:37:49 for the object: Serial-COM1
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

Tips

You should use `instrcallback` as a template from which you create callback functions that suit your specific application needs.

Note Using this callback for numbers greater than 127 with a terminator is not supported.

Introduced before R2006a

instrfind

Read instrument objects from memory to MATLAB workspace

Syntax

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

Arguments

'*PropertyName*'	A property name for `obj`.
`PropertyValue`	A property value supported by *PropertyName*.
`S`	A structure of property names and property values.
`obj`	An instrument object, or an array of instrument objects.
`out`	An array of instrument objects.

Description

`out = instrfind` returns all valid instrument objects as an array to `out`.

`out = instrfind('`*PropertyName*`',PropertyValue,...)` returns an array of instrument objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of instrument objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'`*PropertyName*`',PropertyValue,...)` restricts the search for matching property name/property value pairs to the instrument objects listed in `obj`.

Examples

Suppose you create the following two GPIB objects.

```
g1 = gpib('ni',0,1);
g2 = gpib('ni',0,2);
g2.EOSCharCode = 'CR';
fopen([g1 g2])
```

You can use `instrfind` to return instrument objects based on property values.

```
out1 = instrfind('Type','gpib');
out2 = instrfind({'Type','EOSCharCode'},{'gpib','CR'});
```

You can also use `instrfind` to return cleared instrument objects to the MATLAB workspace.

```
clear g1 g2
newobjs = instrfind

Instrument Object Array
   Index:   Type:          Status:     Name:
   1        gpib           open        GPIB0-1
   2        gpib           open        GPIB0-2
```

Assign the instrument objects their original names.

```
g1 = newobjs(1);
g2 = newobjs(2);
```

Close both `g1` and `g2`.

```
fclose(newobjs)
```

Tips

`instrfind` will not return an instrument object if its `ObjectVisibility` property is configured to `off`.

You must specify property values using the same format property requires. For example, if the `Name` property value is specified as `MyObject`, `instrfind` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of character vector values. For example, `instrfind` will find an object with a `Parity` property value of `Even` or `even`. You can use the `propinfo` function to determine if a property has a finite set of character vector values.

If you want to clear all of the objects that are found with `instrfind`, you can use the `instrreset` function.

You can use property name/property value character vector pairs, structures, and cell array pairs in the same call to `instrfind`.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

See Also

`clear` | `instrfindall` | `propinfo` | ObjectVisibility | `get` | `instrreset`

Introduced before R2006a

instrfindall

Find visible and hidden instrument objects

Syntax

```
out = instrfindall
out = instrfindall('P1',V1,...)
out = instrfindall(s)
out = instrfindall(objs,'P1',V1,...)
```

Arguments

`'P1'`	Name of an instrument object property or device group object property
`V1`	Value allowed for corresponding *P1*.
`s`	A structure of property names and property values.
`objs`	An array of instrument objects or device group objects.
`out`	An array of returned instrument objects or device group objects.

Description

`out = instrfindall` finds all instrument objects and device group objects, regardless of the value of the objects' `ObjectVisibility` property. The object or objects are returned to `out`.

`out = instrfindall('P1',V1,...)` returns an array, `out`, of instrument objects and device group objects whose property names and corresponding property values match those specified as arguments.

`out = instrfindall(s)` returns an array, `out`, of instrument objects whose property names and corresponding property values match those specified in the structure `s`, where the field names correspond to property names and the field values correspond to the current value of the respective property.

`out = instrfindall(objs,'P1',V1,...)` restricts the search for objects with matching property name/value pairs to the instrument objects and device group objects listed in `objs`.

Note that you can use character vector property name/property value pairs, structures, and cell array property name/property value pairs in the same call to `instrfindall`.

Examples

Suppose you create the following instrument objects on a Windows machine.

```
s1 = serial('COM1');
s2 = serial('COM2');
g1 = gpib('mcc',0,2);
g1.ObjectVisibility = 'off'
```

Because object `g1` has its `ObjectVisibility` set to off, it is not visible to commands like `instrfind`:

```
instrfind

    Instrument Object Array
    Index:   Type:          Status:     Name:
    1        serial         closed      Serial-COM1
    2        serial         closed      Serial-COM2
```

However, `instrfindall` finds all objects regardless of the value of `ObjectVisibility`:

```
instrfindall

    Instrument Object Array
    Index:   Type:          Status:     Name:
    1        serial         closed      Serial-COM1
    2        serial         closed      Serial-COM2
    3        gpib           closed      GPIB0-2
```

The following statements use `instrfindall` to return objects with specific property settings, which are passed as cell arrays:

```
props = {'PrimaryAddress','SecondaryAddress};
vals = {2,0};
obj = instrfindall(props,vals);
```

You can use `instrfindall` as an argument when you want to apply the command to all objects, visible and invisible. For example, the following statement makes all objects visible:

```
set(instrfindall,'ObjectVisibility','on')
```

Tips

`instrfindall` differs from `instrfind` in that it finds objects whose `ObjectVisibility` property is set to `off`.

Property values are case sensitive. You must specify property values using the same format as that the property requires. For example, if `Name` property value is specified as `MyObject`, `instrfindall` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of character vector values.

For example, `instrfindall` will find an object with a `Parity` property value of `Even` or `even`. You can use the `propinfo` function to determine if a property has a finite set of character vector values.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

See Also

`instrfind` | `propinfo` | ObjectVisibility

Introduced before R2006a

instrhelp

Help for instrument object type, function, or property

Syntax

```
instrhelp
instrhelp('name')
out = instrhelp('name')
instrhelp(obj)
instrhelp(obj,'name')
out = instrhelp(obj,'name')
```

Arguments

`'name'`	A function name, property name, or instrument object type.
`obj`	An instrument object.
`out`	The help text.

Description

`instrhelp` returns a complete listing of toolbox functions, with a brief description of each.

`instrhelp('name')` returns help for the function, property, or instrument object type specified by *name*.

You can return specific instrument object information by specifying *name* in the form `object/function` or `object.property`. For example, to return the help for a serial port object's `fprintf` function, *name* would be `serial/fprintf`. To return the help for a serial port object's `Parity` property, *name* would be `serial.parity`.

`out = instrhelp('name')` returns the help text to `out`.

`instrhelp(obj)` returns a complete listing of functions and properties for `obj`, with a brief description of each. Help for the constructor is also returned.

`instrhelp(obj,'name')` returns help for the function or property specified by *name* associated with `obj`.

`out = instrhelp(obj,'name')` returns the help text to `out`.

Examples

The following commands illustrate some of the ways you can get function and property help without creating an instrument object:

```
instrhelp gpib
```

```
out = instrhelp('gpib.m');
```

```
instrhelp set
```

```
instrhelp EOSCharCode
```

```
instrhelp('gpib.eoscharcode')
```

The following commands illustrate some of the ways you can get function and property help for an existing instrument object:

```
g = gpib('ni',0,1);
```

```
instrhelp(g)
```

```
instrhelp(g,'EOSMode');
```

```
out = instrhelp(g,'trigger')
```

Tips

When returning property help, the names in the See Also section that contain all uppercase letters are function names. The names that contain a mixture of upper and lowercase letters are property names. When returning function help, the See Also section contains only function names.

You can also display help via the Workspace browser by right-clicking an instrument object, and selecting **Instrument Help** from the context menu.

See Also

`propinfo`

Introduced before R2006a

instrhwinfo

Information about available hardware

Syntax

```
out = instrhwinfo
out = instrhwinfo('interface')
out = instrhwinfo('drivertype')
out = instrhwinfo('interface','adaptor')
out = instrhwinfo('drivertype','drivername')
out = instrhwinfo('ivi','LogicalName')
out = instrhwinfo('interface','adaptor','type')
out = instrhwinfo(obj)
out = instrhwinfo(obj,'FieldName')
```

Arguments

`'interface'`	A supported instrument interface.
`'drivertype'`	Instrument driver type, may be `matlab`, `ivi`, or `vxipnp`.
`'adaptor'`	A supported GPIB or VISA adaptor.
`'drivername'`	Name of `ivi`, VXI*plug&play*, or MATLAB instrument driver.
`'LogicalName'`	IVI logical name value.
`'type'`	Type of VISA interface.
`obj`	An instrument object or array of instrument objects.
`'FieldName'`	A field name or cell array of field names associated with `obj`.
`out`	A structure or array containing hardware information.

Description

`out = instrhwinfo` returns hardware information to the structure `out`. This information includes the toolbox version, the MATLAB software version, and supported interfaces.

`out = instrhwinfo('interface')` returns information related to the interface specified by *`interface`*. *`interface`* can be `Bluetooth`, `gpib`, `i2c`, `serialport`, `spi`, `tcpip`, `udp`, or `visa`. For the GPIB and VISA interfaces, the information includes the installed adaptors. For the serial port interface, the information includes the available ports and the object constructor name. For the TCP/IP and UDP interfaces, the information includes the local host address.

`out = instrhwinfo('drivertype')` returns a structure, `out`, which contains information related to the specified driver type, *`drivertype`*. *`drivertype`* can be `matlab`, `vxipnp`, or `ivi`. If *`drivertype`* is `matlab`, this information includes the MATLAB instrument drivers found on the MATLAB software path. If *`drivertype`* is `vxipnp`, this information includes the found VXI*plug&play* drivers. If *`drivertype`* is `ivi`, this information includes the available logical names and information on the IVI configuration store. You can use an IVI-C driver.

`out = instrhwinfo('interface','adaptor')` returns information related to the adaptor specified by *`adaptor`*, and for the interface specified by *`interface`*. *`interface`* can be `gpib` or

visa. The returned information includes the adaptor version and available hardware. The GPIB adaptors are keysight (note that agilent still also works), ics, mcc, adlink, and ni. The VISA adaptors are keysight (note that agilent still also works), ni, rs, and tek.

out = instrhwinfo('*drivertype*','drivername') returns a structure, out, which contains information related to the specified driver, drivername, for the specified *drivertype*. *drivertype* can be set to matlab, or vxipnp. The available drivername values are returned by out = instrhwinfo('*drivertype*') .

out = instrhwinfo('ivi','LogicalName') returns a structure, out, which contains information related to the specified logical name, LogicalName. The available logical name values are returned by instrhwinfo('ivi').

out = instrhwinfo('*interface*','*adaptor*','*type*') returns a structure, out, which contains information on the specified type, *type*. *interface* can only be visa. *adaptor* can be ni, ics, keysight (note that agilent still also works), mcc, adlink, or tek. *type* can be gpib, vxi, gpib-vxi, serial, or rsib.

out = instrhwinfo(obj) returns information on the adaptor and vendor-supplied DLL associated with the VISA or GPIB object obj. If obj is a serial port, TCPIP, or UDP object, then JAR file information is returned. If obj is an array of instrument objects, then out is a 1-by-n cell array of structures where n is the length of obj.

out = instrhwinfo(obj,'FieldName') returns hardware information for the field name specified by FieldName. FieldName can be a single character vector or a cell array of character vectors. out is an m-by-n cell array where m is the length of obj and n is the length of FieldName. You can return the supported values for FieldName using the instrhwinfo(obj) syntax.

Examples

The following commands illustrate some of the ways you can get hardware-related information without creating an instrument object.

```
out1 = instrhwinfo;
out2 = instrhwinfo('serialport');
out3 = instrhwinfo('gpib','ni');
out4 = instrhwinfo('visa','agilent');
```

The following commands illustrate some of the ways you can get hardware-related information for an existing instrument object.

```
vs = visa('agilent','ASRL1::INSTR');
out5 = instrhwinfo(vs)
out5 =
           AdaptorDllName: [1x67 char]
        AdaptorDllVersion: 'Version 1.2 (R13)'
              AdaptorName: 'AGILENT'
  VendorDriverDescription: 'Agilent Technologies VISA Driver'
      VendorDriverVersion: '1.1000'

vsdll = instrhwinfo(vs,'AdaptorDllName')
vsdll = D:\V6\toolbox\instrument\instrumentadaptors\win32\
mwagilentvisa.dll
```

Tips

You can also display hardware information via the Workspace browser by right-clicking an instrument object, and selecting **Display Hardware Info** from the context menu.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Introduced before R2006a

instrid

Define and retrieve commands that identify instruments

Syntax

```
instrid
instrid('cmd')
out = instrid(...)
```

Arguments

`cmd`	The instrument identification command.
`out`	The list of commands used to locate and identify instruments.

Description

`instrid` returns the currently defined instrument identification commands.

`instrid('cmd')` defines the instruments identification commands to be the string `cmd`. Note that you can also specify a cell array of commands.

`out = instrid(...)` returns the instrument identification commands to `out`.

Examples

Set the identification command to `*ID?`.

```
instrid('*ID?')
```

Specify three new identification commands using a cell array.

```
instrid({'*IDN?','*ID?','IDEN?'})
```

Assign a list of current identification commands to an output variable.

```
id_commands = instrid;
```

Tips

The Instrument Control Toolbox `instrhwinfo` and `tmtool` functions use the instrument identification commands as defined by `instrid` when locating and identifying instruments.

By default, Instrument Control Toolbox software uses the command `*IDN?`, which identifies most instruments. However, some instruments respond to different identification commands such as `*ID?` or `*IDEN?`.

If `instrhwinfo` or `tmtool` does not identify a known instrument, use `instrid` to specify the identification commands the instrument will respond to. If `instrid` returns no commands, an instrument cannot be found.

See Also

`instrhwinfo`

Introduced before R2006a

instrnotify

Define notification for instrument events

Syntax

```
instrnotify('Type', callback)
instrnotify({'P1', 'P2', ...}, 'Type', callback)
instrnotify(obj, 'Type', callback)
instrnotify(obj, {'P1', 'P2', ...}, 'Type', callback)
instrnotify('Type', callback, '-remove')
instrnotify(obj, 'Type', callback, '-remove')
```

Arguments

`'Type'`	The type of event: `ObjectCreated`, `ObjectDeleted`, or `PropertyChangedPostSet`
`callback`	Function handle, character vector, or cell array to evaluate.
`'P1', P2', ...`	Any number of object property names.
`obj`	Instrument object or device group object.
`'-remove'`	Argument to remove specified callback.

Description

`instrnotify('Type', callback)` evaluates the MATLAB expression, `callback`, in the MATLAB workspace when an event of type *Type* is generated. *Type* can be `ObjectCreated`, `ObjectDeleted`, or `PropertyChangedPostSet`.

If *Type* is `ObjectCreated`, `callback` is evaluated each time an instrument object or a device group object is created. If *Type* is `ObjectDeleted`, `callback` is evaluated each time an instrument object or a device group object is deleted. If *Type* is `PropertyChangedPostSet`, `callback` is evaluated each time an instrument object or device group object property is configured with `set`.

`callback` can be

- A function handle
- A character vector to be evaluated
- A cell array containing the function to evaluate in the first cell (function handle or name of function) and extra arguments to pass to the function in subsequent cells

The `callback` function is invoked with

```
function(obj, event, [arg1, arg2,...])
```

where `obj` is the instrument object or device group object generating the event. `event` is a structure containing information on the event generated. If *Type* is `ObjectCreated` or `ObjectDeleted`, `event` contains the type of event. If *Type* is `PropertyChangedPostSet`, `event` contains the type of event, the property being configured, and the new property value.

`instrnotify({'P1', 'P2', ...}, 'Type', callback)` evaluates the MATLAB expression, `callback`, in the MATLAB workspace when any of the specified properties, *P1*, *P2*, ... are configured. *Type* can be only `PropertyChangedPostSet`.

`instrnotify(obj, 'Type', callback)` evaluates the MATLAB expression, `callback`, in the MATLAB workspace when an event of type *Type* for object `obj`, is generated. `obj` can be an array of instrument objects or device group objects.

`instrnotify(obj, {'P1', 'P2', ...}, 'Type', callback)` evaluates the MATLAB expression, `callback`, in the MATLAB workspace when any of the specified properties, *P1*, *P2*, are configured on object `obj`.

`instrnotify('Type', callback, '-remove')` removes the specified `callback` of type *Type*.

`instrnotify(obj, 'Type', callback, '-remove')` removes the specified `callback` of type *Type* for object `obj`.

Examples

```
instrnotify('PropertyChangedPostSet', @instrcallback);
g = gpib('mcc', 0, 5);
set(g, 'Name', 'mygpib');
fopen(g);
fclose(g);
instrnotify('PropertyChangedPostSet',@instrcallback,'-remove');
```

Tips

`PropertyChangedPostSet` events are generated only when the property is configured to a different value than what the property is currently configured to. For example, if a GPIB object's `Tag` property is configured to `'myobject'`, a `PropertyChangedPostSet` event will not be generated if the object's `Tag` property is currently set to `'myobject'`. A `PropertyChangedPostSet` event will be generated if the object's `Tag` property is set to `'myGPIBObject'`.

If `obj` is specified and the callback *Type* is `ObjectCreated`, the callback will not be generated because `obj` has already been created.

If *Type* is `ObjectDeleted`, the invalid object `obj` is not passed as the first input argument to the `callback` function. Instead, an empty matrix is passed as the first input argument.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Introduced before R2006a

instrreset

Disconnect and delete all instrument objects

Syntax

```
instrreset
```

Description

`instrreset` disconnects and deletes all instrument objects.

Tips

If data is being written or read asynchronously, the asynchronous operation is stopped.

`instrreset` is equivalent to issuing the `stopasync` (if needed), `fclose`, and `delete` functions for all instrument objects.

When you delete an instrument object, it becomes *invalid*. Because you cannot connect an invalid object to the instrument, you should remove it from the workspace with the `clear` command.

See Also

`clear` | `delete` | `fclose` | `isvalid` | `stopasync`

Introduced before R2006a

invoke

Execute driver-specific function on device object

Syntax

```
out = invoke(obj,'name')
out = invoke(obj,'name',arg1,arg2,...)
```

Arguments

`obj`	A device object.
`name`	The function to execute.
`arg1,arg2,...`	Arguments passed to `name`.
`out`	The function output.

Description

`out = invoke(obj,'name')` executes the function specified by `name` on the device object specified by `obj`. The function's output is returned to `out`.

`out = invoke(obj,'name',arg1,arg2,...)` passes the arguments `arg1,arg2,...` to the function specified by `name`.

Examples

Create a device object for a Tektronix TDS 210 oscilloscope that is connected to a National Instruments GPIB board.

```
g = gpib('ni',0,2);
d = icdevice('tektronix_tds210',g);
```

Perform a self-calibration for the oscilloscope by invoking the `calibrate` function.

```
out = invoke(d,'calibrate')
out =
    '0'
```

`0` indicates that the self-calibration completed without any errors.

Tips

To list the driver-specific functions supported by `obj`, type

```
methods(obj)
```

To display help for a specific function, type

```
instrhelp(obj,'name')
```

See Also

`instrhelp` | `methods` | Status

Introduced before R2006a

isvalid

Determine whether instrument objects are valid

Syntax

`out = isvalid(obj)`

Arguments

`obj`	An instrument object or array of instrument objects.
`out`	A logical array.

Description

`out = isvalid(obj)` returns the logical array `out`, which contains a 0 where the elements of `obj` are invalid instrument objects and a 1 where the elements of `obj` are valid instrument objects.

Examples

Suppose you create the following two GPIB objects:

```
g1 = gpib('ni',0,1);
g2 = gpib('ni',0,2);
```

`g2` becomes invalid after it is deleted.

```
delete(g2)
```

`isvalid` verifies that `g1` is valid and `g2` is invalid.

```
garray = [g1 g2];
isvalid(garray)
ans =
     1     0
```

Tips

`obj` becomes invalid after it is removed from memory with the `delete` function. Because you cannot connect an invalid object to the instrument, you should remove it from the workspace with the `clear` command.

See Also

`clear` | `delete`

Introduced before R2006a

iviconfigurationstore

Create IVI configuration store object

Syntax

```
obj = iviconfigurationstore
obj = iviconfigurationstore('file')
```

Arguments

`obj`	IVI configuration store object
`'file'`	Configuration store data file

Description

`obj = iviconfigurationstore` creates an IVI configuration store object and establishes a connection to the IVI Configuration Server. The data in the master configuration store is used.

`obj = iviconfigurationstore('file')` creates an IVI configuration store object and establishes a connection to the IVI Configuration Server. The data in the configuration store, `file`, is used. If `file` cannot be found or is not a valid configuration store, an error occurs.

See Also

`add` | `commit` | `remove` | `update`

Introduced before R2006a

length

Length of instrument object array

Syntax

```
length(obj)
```

Arguments

`obj`	An instrument object or an array of instrument objects.

Description

`length(obj)` returns the length of `obj`. It is equivalent to the command `max(size(obj))`.

See Also

`instrhelp` | `size`

Introduced before R2006a

load

Load instrument objects and variables into MATLAB workspace

Syntax

```
load filename
load filename obj1 obj2 ...
out = load('filename','obj1','obj2',...)
```

Arguments

`filename`	The MAT-file name.
`obj1 obj2 ...`	Instrument objects or arrays of instrument objects.
`out`	A structure containing the specified instrument objects.

Description

`load filename` returns all variables from the MAT-file specified by `filename` into the MATLAB workspace.

`load filename obj1 obj2 ...` returns the instrument objects specified by `obj1 obj2...` from the MAT-file `filename` into the MATLAB workspace.

`out = load('filename','obj1','obj2',...)` returns the specified instrument objects from the MAT-file `filename` as a structure to `out` instead of directly loading them into the workspace. The field names in `out` match the names of the loaded instrument objects.

Examples

Suppose you create the GPIB objects `g1` and `g2`, configure a few properties for `g1`, and connect both objects to their associated instruments.

```
g1 = gpib('ni',0,1);
g2 = gpib('ni',0,2);
set(g1,'EOSMode','read','EOSCharCode','CR')
fopen([g1 g2])
```

The read-only `Status` property is automatically configured to `open`.

```
g1.Status
ans =
    open

g2.Status
ans =
    open
```

Save `g1` and `g2` to the file `MyObject.mat`, and then load the objects into the MATLAB workspace.

```
save MyObject g1 g2
load MyObject g1 g2
```

Values for read-only properties are restored to their default values upon loading, while all other property values are honored.

```
get([g1 g2],{'EOSMode','EOSCharCode','Status'})
ans =
    'read'     'CR'     'closed'
    'none'     'LF'     'closed'
```

Tips

Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages or use the `propinfo` function.

See Also

`instrhelp` | `propinfo` | `save`

Introduced before R2006a

makemid

Convert driver to MATLAB instrument driver format

Syntax

```
makemid('driver')
makemid('driver', 'filename')
makemid('driver', 'type')
makemid('driver', 'filename', 'type')
```

Arguments

`'driver'`	Name of driver being converted.
`'filename'`	Name of file that the converted driver is saved to. You may specify a full pathname. If an extension is not specified, the `.mdd` extension is used.
`'type'`	The type of driver the function looks for. By default, the function searches among all types.

Description

`makemid('driver')` searches through known driver types for `driver` and creates a MATLAB instrument driver representation of the driver. Known driver types include VXI*plug&play* and IVI-C. For `driver` you can use a `Module` (for IVI-C), a `LogicalName` (for IVI-C), or the original VXI*plug&play* instrument driver name. The MATLAB instrument driver will be saved in the current working directory as `driver.mdd`

The MATLAB instrument driver can then be modified using `midedit` to customize the driver behavior, and may be used to instantiate a device object using `icdevice`.

`makemid('driver', 'filename')` creates and saves the MATLAB instrument driver using the name and path specified by `filename`.

`makemid('driver', 'type')` and `makemid('driver', 'filename', 'type')` override the default search order and look only for drivers whose type is *`type`*. Valid types are `vxiplug&play` and `ivi-c`.

The function searches for the specified driver root interface. For example, if the driver supports the `IIviScope` interface, an `interface` value of `IIviScope` results in a device object that only contains the `IVIScope` class-compliant properties and methods.

Note MAKEMID is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

Examples

To convert the driver `hp34401` into the MATLAB instrument driver `hp34401.mdd` in the current working directory,

```
makemid('hp34401');
```

To convert the driver `tktds5k` into the MATLAB instrument driver with a specific name and location,

```
makemid('tktds5k', 'C:\MyDrivers\tektronix_5k.mdd');
```

To convert the IVI-C driver `tktds5k` into the MATLAB instrument driver `tktds5k.mdd` in the current working directory. This example causes the function to look for the driver only among the IVI-C drivers.

```
makemid('tktds5k', 'ivi-c');
```

To create the MATLAB instrument driver `MyIviLogicalName.mdd` from the IVI logical name `MyIviLogicalName`,

```
makemid('MyIviLogicalName');
```

See Also

`icdevice` | `midedit`

Introduced before R2006a

maskWrite

Perform mask write operation on a holding register

Syntax

```
maskWrite(m,address,andMask,orMask)
maskWrite(m,address,andMask,orMask,serverId)
```

Description

`maskWrite(m,address,andMask,orMask)` writes data to MODBUS object `m` to a holding register at address `address`, using the indicated mask values. The function can set or clear individual bits in a specific holding register. It is a read/modify/write operation, and uses a combination of an AND mask, an OR mask, and the current contents of the register.

`maskWrite(m,address,andMask,orMask,serverId)` additionally specifies the `serverId` as the address of the server to send the write command to.

Examples

Perform a Mask Read on a Holding Register

You can modify the contents of a holding register using the `maskWrite` function. The function can set or clear individual bits in a specific holding register. It is a read/modify/write operation, and uses a combination of an AND mask, an OR mask, and the current contents of the register.

Create the `AND` and `OR` variables.

```
andMask = 6
orMask = 0
```

Set bit 0 at address 20, and perform a mask write operation. Since the `andMask` is a 6, that clears all bits except for bits 1 and 2. Bits 1 and 2 are preserved.

```
maskWrite(m,20,andMask,orMask)
```

Perform a Mask Read on a Holding Register, and Specify Server ID

Use the `serverId` argument to specify the address of the server to send the mask write command to.

Set bit 0 at address 20 and perform a mask write operation at server ID 3.

```
maskWrite(m,20,6,0,3)
```

Input Arguments

`address` — Register address to perform mask write operation on
double

Register address to perform mask write operation on, specified as a double. Address must be the first argument after the object name. This example sets bit 0 at address 20 and performs a mask write operation.

Example: `maskWrite(m,20,andMask,orMask)`

Data Types: `double`

`andMask` — AND value to use in mask write operation
double

AND value to use in mask write operation, specified as a double. `andMask` must be the second argument after the object name. The valid range is `0–65535`.

This example sets bit 0 at address 20 and performs a mask write operation, using 6 as the AND value.

Example: `maskWrite(m,20,6,0)`

Data Types: `double`

`orMask` — OR value to use in mask write operation
double

OR value to use in mask write operation, specified as a double. `orMask` must be the third argument after the object name. The valid range is `0–65535`.

This example sets bit 0 at address 20 and performs a mask write operation, using 0 as the OR value.

Example: `maskWrite(m,20,6,0)`

Data Types: `double`

`serverId` — Address of the server to send the mask write command to
double

Address of the server to send the mask write command to, specified as a double. Server ID must be specified after the object name, address, AND mask, and OR mask. If you do not specify a `serverId`, the default of `1` is used. Valid values are `0–247`, with `0` being the broadcast address. This example sets bit 0 at address 20 and performs a mask write operation at server ID 3.

Example: `maskWrite(m,20,6,0,3)`

Data Types: `double`

Tips

The function algorithm works as follows:

```
Result = (register value AND andMask) OR (orMask AND (NOT andMask))
```

For example:

```
                 Hex    Binary
Current contents  12   0001 0010
And_Mask          F2   1111 0010
Or_Mask           25   0010 0101
(NOT And_Mask)    0D   0000 1101

Result            17   0001 0111
```

If the `orMask` value is 0, the result is simply the logical ANDing of the current contents and the `andMask`. If the `andMask` value is 0, the result is equal to the `orMask` value.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`modbus` | `read` | `write` | `writeRead`

Topics
"Create a MODBUS Connection" on page 11-3
"Configure Properties for MODBUS Communication" on page 11-5
"Modify the Contents of a Holding Register Using a Mask Write" on page 11-18

Introduced in R2017a

memmap

(To be removed) Map memory for low-level memory read and write operations

Note This `visa` object function will be removed in a future release. Use `visadev` object functions instead. For more information, see "Compatibility Considerations".

Syntax

`memmap(obj,'adrspace',offset,size)`

Arguments

`obj`	A VISA-VXI or VISA-GPIB-VXI object.
`'adrspace'`	The memory address space.
`offset`	Offset for the memory address space.
`size`	Number of bytes to map.

Description

`memmap(obj,'adrspace',offset,size)` maps the amount of memory specified by `size` in address space, *`adrspace`* with an offset, `offset`. You can configure *`adrspace`* to `A16` (A16 address space), `A24` (A24 address space), or `A32` (A32 address space).

Examples

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

Use `memmap` to map 16 bytes in the A16 address space.

```
memmap(vv,'A16',0,16)
```

Read the first and second instrument registers.

```
reg1 = mempeek(vv,0,'uint16');
reg2 = mempeek(vv,2,'uint16');
```

Unmap the memory and disconnect `vv` from the instrument.

```
memunmap(vv)
fclose(vv)
```

Tips

Before you can map memory, `obj` must be connected to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to map memory while `obj` is not connected to the instrument.

To unmap the memory, use the `memunmap` function. If memory is mapped and `fclose` is called, the memory is unmapped before the object is disconnected from the instrument.

The `MappedMemorySize` property returns the size of the memory space mapped. You must map the memory space before using the `mempoke` or `mempeek` function.

Compatibility Considerations

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

See Also
`fopen` | `fclose` | `mempeek` | `mempoke` | `memunmap` | MappedMemorySize | Status

Introduced before R2006a

mempeek

(To be removed) Low-level memory read from VXI register

Note This `visa` object function will be removed in a future release. Use `visadev` object functions instead. For more information, see "Compatibility Considerations".

Syntax

```
out = mempeek(obj,offset)
out = mempeek(obj,offset,'precision')
```

Arguments

`obj`	A VISA-VXI or VISA-GPIB-VXI object.
`offset`	The offset in the mapped memory space from which the data is read.
`'`*`precision`*`'`	The number of bits to read from the memory address.
`out`	An array containing the returned value.

Description

`out = mempeek(obj,offset)` reads a `uint8` value from the mapped memory space specified by `offset` for the object `obj`. The value is returned to `out`.

`out = mempeek(obj,offset,'`*`precision`*`')` reads the number of bits specified by *`precision`*, from the mapped memory space specified by `offset`. *`precision`* can be `uint8`, `uint16`, or `uint32`, which instructs `mempeek` to read 8-, 16-, or 32-bit values, respectively. *`precision`* can also be `single`, which instructs `mempeek` to read single precision values.

Examples

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

Use `memmap` to map 16 bytes in the A16 address space.

```
memmap(vv,'A16',0,16)
```

Perform a low-level read of the first and second instrument registers.

```
reg1 = mempeek(vv,0,'uint16')
reg1 =
       53247
reg2 = mempeek(vv,2,'uint16')
reg2 =
       20993
```

Unmap the memory and disconnect `vv` from the instrument.

```
memunmap(vv)
fclose(vv)
```

Tips

Before you can read from the VXI register, `obj` must be connected to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt a read operation while `obj` is not connected to the instrument.

You must map the memory space using the `memmap` function before using `mempeek`. The `MappedMemorySize` property returns the size of the memory space mapped.

`offset` indicates the offset in the mapped memory space from which the data is read. For example, if the mapped memory space begins at 200H, the offset is 2, and the precision is `uint8`, then the data is read from memory location 202H. If the precision is `uint16`, the data is read from 202H and 203H.

To increase speed, `mempeek` does not return error messages from the instrument.

Compatibility Considerations

`visa` object interface will be removed

Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

See Also

`fopen` | `memmap` | `mempoke` | `memunmap` | MappedMemorySize | MemoryIncrement | Status

Introduced before R2006a

mempoke

(To be removed) Low-level memory write to VXI register

Note This `visa` object function will be removed in a future release. Use `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
mempoke(obj,data,offset)
mempoke(obj,data,offset,'precision')
```

Arguments

`obj`	A VISA-VXI or VISA-GPIB-VXI object.
`data`	The data written to the memory address.
`offset`	The offset in the mapped memory space to which the data is written.
`'precision'`	The number of bits to write to the memory address.

Description

`mempoke(obj,data,offset)` writes the `uint8` value specified by `data` to the mapped memory address specified by `offset` for the object `obj`.

`mempoke(obj,data,offset,'precision')` writes `data` using the number of bits specified by *precision*. *precision* can be `uint8`, `uint16`, or `uint32`, which instructs `mempoke` to write `data` as 8-, 16-, or 32-bit values, respectively. *precision* can also be `single`, which instructs `mempoke` to write `data` as single-precision values.

Examples

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

Use `memmap` to map 16 bytes in the A16 address space.

```
memmap(vv,'A16',0,16)
```

Perform a low-level write to the fourth instrument register, which has an offset of 6.

```
mempoke(vv,45056,6,'uint16')
```

Unmap the memory and disconnect `vv` from the instrument.

```
memunmap(vv)
fclose(vv)
```

Tips

Before you can write to the VXI register, `obj` must be connected to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt a write operation while `obj` is not connected to the instrument.

You must map the memory space using the `memmap` function before using `mempoke`. The `MappedMemorySize` property returns the size of the memory space mapped.

`offset` indicates the offset in the mapped memory space to which the data is written. For example, if the mapped memory space begins at 200H, the offset is 2, and the precision is `uint8`, then the data is written to memory location 202H. If the precision is `uint16`, the data is written to 202H and 203H.

To increase speed, `mempoke` does not return error messages from the instrument.

Compatibility Considerations

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

See Also

`fopen` | `memmap` | `mempeek` | MappedMemorySize | MemoryIncrement | Status

Introduced before R2006a

memread

(To be removed) High-level memory read from VXI register

Note This `visa` object function will be removed in a future release. Use `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
out = memread(obj)
out = memread(obj,offset)
out = memread(obj,offset,'precision')
out = memread(obj,offset,'precision','adrspace')
out = memread(obj,offset,'precision','adrspace',size)
```

Arguments

`obj`	A VISA-VXI or VISA-GPIB-VXI object.
`offset`	Offset for the memory address space.
`'`*`precision`*`'`	The number of bits to read from the memory address.
`'`*`adrspace`*`'`	The memory address space.
`offset`	Offset for the memory address space.
`size`	The size of the data block to read.
`out`	An array containing the returned value.

Description

`out = memread(obj)` reads a `uint8` value from the `A16` address space with an offset of 0 for the object `obj`.

`out = memread(obj,offset)` reads a `uint8` value from the A16 address space with an offset specified by `offset`. You must specify `offset` as a decimal value.

`out = memread(obj,offset,'`*`precision`*`')` reads the number of bits specified by *`precision`* from the `A16` address space. *`precision`* can be `uint8`, `uint16`, or `uint32`, which instructs `memread` to read 8-, 16-, or 32-bit values, respectively. *`precision`* can also be `single`, which instructs `memread` to read single-precision values.

`out = memread(obj,offset,'`*`precision`*`','`*`adrspace`*`')` reads the specified number of bits from the address space specified by *`adrspace`*. *`adrspace`* can be `A16`, `A24`, or `A32`. The `MemorySpace` property indicates which VXI address spaces are used by the instrument.

`out = memread(obj,offset,'`*`precision`*`','`*`adrspace`*`',size)` reads a block of data with a size specified by `size`.

Examples

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

Perform a high-level read of the first instrument register.

```
reg1 = memread(vv,0,'uint16')
reg1 =
       53247
```

Perform a high-level read of the next three instrument registers.

```
reg24 = memread(vv,2,'uint16','A16',3)
reg24 =
       20993
       50012
       40960
```

Disconnect `vv` from the instrument.

```
fclose(vv)
```

Tips

Before you can read data from the VXI register, `obj` must be connected to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to read memory while `obj` is not connected to the instrument.

Compatibility Considerations

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

See Also

`fopen` | `mempeek` | `memwrite` | MemoryIncrement | MemorySpace | Status

Introduced before R2006a

memunmap

(To be removed) Unmap memory for low-level memory read and write operations

Note This `visa` object function will be removed in a future release. Use `visadev` object functions instead. For more information, see "Compatibility Considerations".

Syntax

`memunmap(obj)`

Arguments

`obj`	A VISA-VXI or VISA-GPIB-VXI object.

Description

`memunmap(obj)` unmaps memory space previously mapped by the `memmap` function.

Examples

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

Map 16 bytes in the A16 address space.

```
memmap(vv,'A16',0,16)
```

Read the first and second instrument registers.

```
reg1 = mempeek(vv,0,'uint16');
reg2 = mempeek(vv,2,'uint16');
```

Use `memunmap` to unmap the memory, and disconnect `vv` from the instrument.

```
memunmap(vv)
fclose(vv)
```

Tips

When the memory space is unmapped, the `MappedMemorySize` property is set to `0` and the `MappedMemoryBase` property is set to `0H`.

Compatibility Considerations

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

See Also

`memmap` | `mempeek` | `mempoke` | MappedMemoryBase | MappedMemorySize

Introduced before R2006a

memwrite

(To be removed) High-level memory write to VXI register

Note This `visa` object function will be removed in a future release. Use `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
memwrite(obj,data)
memwrite(obj,data,offset)
memwrite(obj,data,offset,'precision')
memwrite(obj,data,offset,'precision','adrspace')
```

Arguments

`obj`	A VISA-VXI or VISA-GPIB-VXI object.
`data`	The data written to the memory address.
`offset`	Offset for the memory address space.
`'precision'`	The number of bits to write to the memory address.
`'adrspace'`	The memory address space.

Description

`memwrite(obj,data)` writes the `uint8` value specified by `data` to the A16 address space with an offset of 0 for the object `obj`. `data` can be an array of `uint8` values.

`memwrite(obj,data,offset)` writes `data` to the A16 address space with an offset specified by `offset`. `offset` is specified as a decimal value.

`memwrite(obj,data,offset,'precision')` writes `data` with precision specified by *`precision`*. *`precision`* can be `uint8`, `uint16`, or `uint32`, which instructs `memwrite` to write `data` as 8-, 16-, or 32-bit values, respectively. *`precision`* can also be `single`, which instructs `memwrite` to write `data` as single-precision values.

`memwrite(obj,data,offset,'precision','adrspace')` writes `data` to the address space specified by *`adrspace`*. *`adrspace`* can be `A16`, `A24`, or `A32`. The `MemorySpace` property indicates which VXI address spaces are used by the instrument.

Examples

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

Perform a high-level write to the fourth instrument register, which has an offset of 6.

```
memwrite(vv,45056,6,'uint16','A16')
```

Disconnect vv from the instrument.

```
fclose(vv)
```

Tips

Before you can write to the VXI register, obj must be connected to the instrument with the fopen function. A connected interface object has a Status property value of open. An error is returned if you attempt a write operation while obj is not connected to the instrument.

Compatibility Considerations

visa object interface will be removed
Not recommended starting in R2021a

Use of this function with a visa object will be removed. To access a VISA resource, use a visadev object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

See Also

fopen | memread | mempoke | MemoryIncrement | MemorySpace | Status

Introduced before R2006a

methods

Class method names and descriptions

Syntax

```
m = methods('classname')
m = methods(object)
m = methods(...,'-full')
```

Arguments

`m`	Cell array of character vectors
`'classname'`	Class whose methods are returned
`object`	An instrument object or device group object
`'-full'`	Request to return full descriptions of methods

Description

`m = methods('classname')` returns, in a cell array of character vectors, the names of all methods for the class with the name `classname`.

`m = methods(object)` returns the names of all methods for the class of which `object` is an instance.

`m = methods(...,'-full')` returns full descriptions of the methods in the class, including inheritance information and, for Java® methods, also attributes and signatures. Duplicate method names with different signatures are not removed. If `classname` represents a MATLAB class, then inheritance information is returned only if that class has been instantiated.

Tips

`methods` differs from `what` in that the methods from all method directories are reported together, and `methods` removes all duplicate method names from the result list. `methods` will also return the methods for a Java class.

See Also

`methodsview` | `what` | `which` | `help`

Introduced before R2006a

midedit

Open graphical tool for creating and editing MATLAB instrument driver

Syntax

```
midedit
midedit('driver')
```

Arguments

`'driver'`	The name of a MATLAB instrument driver.

Description

`midedit` opens the MATLAB Instrument Driver Editor, which is a graphical tool for creating and editing instrument drivers.

`midedit('driver')` opens the MATLAB Instrument Driver Editor for the specified instrument driver. The default extension for driver is `.mdd`. Note that `driver` can include a relative partial pathname.

The editor consists of two main parts: the navigation pane and the detail pane. The navigation pane lists the driver-specific properties and functions in a tree view, while the detail pane allows you to configure and document the properties and functions.

`midedit` may also be used to import VXI*plug&play* or IVI drivers. With `midedit` open, select **Import** from the **File** menu. The import process creates a new MATLAB Instrument Driver based on the VXI*plug&play* or IVI driver. This allows you to customize the behavior of device objects that use the VXI*plug&play* or IVI driver.

For details and examples on the MATLAB Instrument Driver Editor, see "MATLAB Instrument Driver Editor Overview" on page 19-2.

Note MIDEDIT is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

See Also

`icdevice` | `makemid` | `midtest`

Introduced before R2006a

midtest

Open graphical tool for testing MATLAB instrument driver

Syntax

```
midtest
midtest('file')
```

Arguments

`'file'`	File containing the test to be used by the MATLAB Instrument Driver Testing Tool

Description

`midtest` opens the MATLAB Instrument Driver Testing Tool. The MATLAB Instrument Driver Testing Tool provides a graphical environment for creating a test to verify the functionality of a MATLAB instrument driver.

The MATLAB Instrument Driver Testing Tool provides a way to

- Verify property behavior
- Verify function behavior
- Save the test as MATLAB code
- Export the test results to MATLAB workspace, figure window, MAT-file, or the MATLAB Variables editor
- Save test results as an HTML page

`midtest('file')` opens the MATLAB Instrument Driver Testing Tool with the test loaded from `file`.

For a full description of the tool with examples, see “Instrument Driver Testing Tool Overview” on page 20-2.

Examples

```
midtest('test.xml')
```

opens the MATLAB Instrument Driver Testing Tool with the test `test.xml` loaded.

See Also

`icdevice` | `makemid` | `midedit`

Introduced before R2006a

modbus

Create MODBUS object

Syntax

```
m = modbus(Transport,DeviceAddress)
m = modbus(Transport,DeviceAddress,Port)
m = modbus(Transport,DeviceAddress,Name,Value)
m = modbus(Transport,'Port')
m = modbus(Transport,'Port',Name,Value)
```

Description

`m = modbus(Transport,DeviceAddress)` constructs a MODBUS object, `m`, over the transport type `Transport` using the specified `'DeviceAddress'`. When the transport is `'tcpip'`, `DeviceAddress` must be specified as the second argument. `DeviceAddress` is the IP address or host name of the MODBUS server.

`m = modbus(Transport,DeviceAddress,Port)` additionally specifies `Port`. When the transport is `'tcpip'`, `DeviceAddress` must be specified. `Port` is the remote port used by the MODBUS server. Port is optional, and it defaults to 502, which is the reserved port for MODBUS.

`m = modbus(Transport,DeviceAddress,Name,Value)` specifies additional options with one or more name-value pair arguments using any of the previous syntaxes. For example, you can specify a timeout value. The `Timeout` property specifies the waiting time to complete read and write operations in seconds, and the default is `10`.

`m = modbus(Transport,'Port')` constructs a MODBUS object `m` over the transport type `Transport` using the specified `'Port'`. When the transport is `'serialrtu'`, `'Port'` must be specified. This argument is the serial port the MODBUS server is connected to, such as `'COM3'`.

`m = modbus(Transport,'Port',Name,Value)` specifies additional options with one or more name-value pair arguments using any of the previous syntaxes. For example, you can specify `NumRetries`, the number of retries to perform if there is no reply from the server after a timeout.

Examples

Create Object Using TCP/IP Transport

When the transport is TCP/IP, you must specify the IP address or host name of the MODBUS server. You can optionally specify the remote port used by the MODBUS server. Port defaults to 502, which is the reserved port for MODBUS.

Create the MODBUS object `m` using the host address shown and port of `308`.

```
m = modbus('tcpip', '192.168.2.1', 308)
```

```
m =
```

```
  Modbus TCPIP with properties:

   DeviceAddress: '192.168.2.1'
            Port: 308
          Status: 'open'
      NumRetries: 1
         Timeout: 10 (seconds)
       ByteOrder: 'big-endian'
       WordOrder: 'big-endian'
```

The object output shows both the arguments you set and the defaults.

Create Object Using Serial RTU Transport

When the transport is `'serialrtu'`, you must specify `'Port'`. This is the serial port the MODBUS server is connected to.

Create the MODBUS object `m` using the `Port` of `'COM3'`.

```
m = modbus('serialrtu','COM3')
```

```
m = 

Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 10 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

The object output shows arguments you set and defaults that are used automatically.

Create Object and Set a Property

You can create the object using a name-value pair to set the properties such as `Timeout`. The `Timeout` property specifies the maximum time in seconds to wait for a response from the MODBUS server, and the default is `10`. You can change the value either during object creation or after you create the object.

For the list and description of properties you can set for both transport types, see "Configure Properties for MODBUS Communication."

Create a MODBUS object using Serial RTU, but increase the `Timeout` to `20` seconds.

```
m = modbus('serialrtu','COM3','Timeout',20)
```

```
m =

Modbus Serial RTU with properties:

             Port: 'COM3'
         BaudRate: 9600
         DataBits: 8
           Parity: 'none'
         StopBits: 1
           Status: 'open'
       NumRetries: 1
          Timeout: 20 (seconds)
        ByteOrder: 'big-endian'
        WordOrder: 'big-endian'
```

The object output reflects the `Timeout` property change.

Input Arguments

`Transport` — Physical transport layer for device communication
character vector | string

Physical transport layer for device communication, specified as a character vector or string. Specify transport type as the first argument when you create the `modbus` object. You must set the transport type as either `'tcpip'` or `'serialrtu'` to designate the protocol you want to use.

Example: `m = modbus('tcpip', '192.168.2.1')`

Data Types: `char`

`DeviceAddress` — IP address or host name of MODBUS server
character vector | string

IP address or host name of MODBUS server, specified as a character vector or string. If transport is TCP/IP, it is required as the second argument during object creation.

Example: `m = modbus('tcpip', '192.168.2.1')`

Data Types: `char`

`Port` — Remote port used by MODBUS server
502 (default) | double

Remote port used by MODBUS server, specified as a double. Optional as a third argument during object creation if transport is TCP/IP. The default of 502 is used if none is specified.

Example: `m = modbus('tcpip','192.168.2.1',308)`

Data Types: `double`

`'Port'` — Serial port MODBUS server is connected to
character vector | string

Serial port MODBUS server is connected to, e.g. `'COM1'`, specified as a character vector or string. If transport is Serial RTU, it is required as the second argument during object creation.

Example: `m = modbus('serialrtu','COM3')`

Data Types: `char`

Name-Value Pair Arguments

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

There are a number of name-value pairs that can be used when you create the `modbus` object, including the two shown here. Some can only be used with either TCP/IP or Serial RTU, and some can be used with both transport types. For a list of all the properties and how to set them both during and after object creation, see "Configure Properties for MODBUS Communication" on page 11-5.

Example: `m = modbus('serialrtu','COM3','Timeout',20)`

`Timeout` — Maximum time in seconds to wait for a response from the MODBUS server
10 (default) | double

Maximum time in seconds to wait for a response from the MODBUS server, specified as the comma-separated pair consisting of `'Timeout'` and a positive value of type `double`. The default is `10`. You can change the value either during object creation or after you create the object.

Example: `m = modbus('serialrtu','COM3','Timeout',20)`

Data Types: `double`

`NumRetries` — Number of retries to perform if there is no reply from the server after a timeout
double

Number of retries to perform if there is no reply from the server after a timeout, specified as the comma-separated pair consisting of `'NumRetries'` and a positive value of type `double`. If using the Serial RTU transport, the message is resent. If using the TCP/IP transport, the connection is closed and reopened. You can change the value either during object creation, or after you create the object.

Example: `m = modbus('serialrtu','COM3','NumRetries',5)`

Data Types: `double`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`read` | `write` | `writeRead` | `maskWrite`

Topics

"Create a MODBUS Connection" on page 11-3
"Configure Properties for MODBUS Communication" on page 11-5

Introduced in R2017a

obj2mfile

Convert instrument object to MATLAB code

Syntax

```
obj2mfile(obj,'filename')
obj2mfile(obj,'filename','syntax')
obj2mfile(obj,'filename','mode')
obj2mfile(obj,'filename','syntax','mode')
obj2mfile(obj,'filename','reuse')
obj2mfile(obj,'filename','syntax','mode','reuse')
```

Arguments

`obj`	An instrument object or an array of instrument objects.
`'filename'`	The name of the file that the MATLAB code is written to. You can specify the full pathname. If an extension is not specified, the `.m` extension is used.
`'`*`syntax`*`'`	Syntax of the converted MATLAB code. By default, the `set` syntax is used. If `dot` is specified, then the dot notation is used.
`'`*`mode`*`'`	Specifies whether all properties are converted to code, or only modified properties are converted to code.
`'`*`reuse`*`'`	Specifies whether existing object is reused or new object is created.

Description

`obj2mfile(obj,'filename')` converts `obj` to the equivalent MATLAB code using the `set` syntax and saves the code to `filename`. Only those properties not set to their default value are saved.

`obj2mfile(obj,'filename','`*`syntax`*`')` converts `obj` to the equivalent MATLAB code using the syntax specified by *`syntax`*. You can specify *`syntax`* to be `set` or `dot`. `set` uses the set syntax, while `dot` uses the dot notation.

`obj2mfile(obj,'filename','`*`mode`*`')` converts the properties specified by *`mode`*. You can specify *`mode`* to be `all` or `modified`. If *`mode`* is `all`, then all properties are converted to code. If *`mode`* is `modified`, then only those properties not set to their default value are converted to code.

`obj2mfile(obj,'filename','`*`syntax`*`','`*`mode`*`')` converts the specified properties to code using the specified syntax.

`obj2mfile(obj,'filename','`*`reuse`*`')` check for an existing instrument object, `obj`, before creating `obj`. If *`reuse`* is `reuse`, the object is used if it exists, otherwise the object is created. If *`reuse`* is `create`, the object is always created. By default, *`reuse`* is `reuse`.

An object will be reused if the existing object has the same constructor arguments as the object about to be created, and if their `Type` and `Tag` property values are the same.

`obj2mfile(obj,'filename','`*`syntax`*`','`*`mode`*`','`*`reuse`*`')` check for an existing instrument object, `obj`, before creating `obj`. If *`reuse`* is `reuse`, the object is used if it exists, otherwise the object is created. If *`reuse`* is `create`, the object is always created. By default, *`reuse`* is `reuse`.

An object will be reused if the existing object has the same constructor arguments as the object about to be created, and if their `Type` and `Tag` property values are the same.

Examples

Suppose you create the GPIB object `g`, and configure several property values.

```
g = gpib('ni',0,1);
set(g,'Tag','MyGPIB object','EOSMode','read','EOSCharCode','CR')
set(g,'UserData',{'test',2,magic(10)})
```

The following command writes MATLAB code to the files `MyGPIB.m` and `MyGPIB.mat`.

```
obj2mfile(g,'MyGPIB.m','dot')
```

`MyGPIB.m` contains code that recreates the commands shown above using the dot notation for all properties that have their default values changed. Because `UserData` is set to a cell array of values, this property appears in `MyGPIB.m` as `obj1.UserData = userdata1`.

It is saved in `MyGPIB.mat` as `userdata = {'test', 2, magic(10)}`.

To recreate `g` in the MATLAB workspace using a new variable, `gnew`,

```
gnew = MyGPIB;
```

The associated MAT-file, `MyGPIB.mat`, is automatically run and `UserData` is assigned the appropriate values.

```
gnew.UserData
```

```
ans =

  1×3 cell array

    {'test'}    {[2]}    {10×10 double}
```

Tips

You can recreate a saved instrument object by typing the name of the file at the MATLAB Command Window.

If the `UserData` property is not empty or if any of the callback properties are set to a cell array of values or a function handle, then the data stored in those properties is written to a MAT-file when the instrument object is converted and saved. The MAT-file has the same name as the file containing the instrument object code (see the example below).

Read-only properties are restored with their default values. For example, suppose an instrument object is saved with a `Status` property value of `open`. When the object is recreated, `Status` is set to its default value of `closed`.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For

information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

See Also

`propinfo`

Introduced before R2006a

oscilloscope

Create Quick-Control Oscilloscope object

Syntax

```
myScope = oscilloscope()
connect(myScope);
set(myScope, 'P1',V1,'P2',V2,...)
waveformArray = readWaveform(myScope);
```

Description

The Quick-Control Oscilloscope can be used for any oscilloscope that uses VISA and an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. You can also use it for Tektronix oscilloscopes. This is an easy to use oscilloscope object.

`myScope = oscilloscope()` creates an instance of the scope named `myScope`.

`connect(myScope);` connects to the scope.

`set(myScope, 'P1',V1,'P2',V2,...)` assigns the specified property values.

`waveformArray = readWaveform(myScope);` acquires a waveform from the scope.

For information on the prerequisites for using `oscilloscope`, see "Quick-Control Oscilloscope Requirements" on page 14-21.

The Quick-Control Oscilloscope `oscilloscope` function can use the following special functions, in addition to standard functions such as `connect` and `disconnect`.

Function	Description
`autoSetup`	Automatically configures the instrument based on the input signal. `autoSetup(myScope);`

`disableChannel`	Disables oscilloscope's channels.
	
	`disableChannel('Channel1');`
	`disableChannel({'Channel1','Channel2'});`

`enableChannel`	Enables oscilloscope's channels from which waveforms will be retrieved.
	
	`enableChannel('Channel1');`
	`enableChannel({'Channel1','Channel2'});`

`drivers`	Retrieves a list of available oscilloscope instrument drivers. Returns a list of available drivers with their supported instrument models. `driverlist = drivers(myScope);`

Function	Description
`resources`	Retrieves a list of available resources of instruments. It returns a list of available VISA resource strings when using an IVI-C scope. It returns the interface resource information when using a Tektronix scope. `res = resources(myScope);`

`configureChannel`	Returns or sets specified oscilloscope control on selected channel. Possible controls are:
	• `'VerticalCoupling'`
	• `'VerticalOffset'`
	• `'VeticalRange'`
	• `'ProbeAttenuation'`
	`value = configureChannel(myScope,'Channel1','VerticalOffset');`
	`configureChannel(myScope,'Channel1','VerticalCoupling','AC');`

`getVerticalCoupling`	Returns the value of how the oscilloscope couples the input signal for the selected channel name as a MATLAB character vector. Possible values returned are `'AC'`, `'DC'`, and `'GND'`. `VC = getVerticalCoupling(myScope, 'Channel1');`
`getVerticalOffset`	Returns location of the center of the range for the selected channel name as a MATLAB character vector. The units are volts. `VO = getVerticalOffset(myScope, 'Channel1');`
`getVerticalRange`	Returns absolute value of the input range the oscilloscope can acquire for selected channel name as a MATLAB character vector. The units are volts. `VR = getVerticalRange(myScope, 'Channel1');`
`readWaveform`	Returns the waveforms displayed on the scope screen. Retrieves the waveforms from enabled channels. `w = readWaveform(myScope);`
`setVerticalCoupling`	Specifies how the oscilloscope couples the input signal for the selected channel name as a MATLAB character vector. Valid values are `'AC'`, `'DC'`, and `'GND'`. `setVerticalCoupling(myScope, 'Channel1', 'AC');`
`setVerticalOffset`	Specifies location of the center of the range for the selected channel name as a MATLAB character vector. For example, to acquire a sine wave that spans between 0.0 and 10.0 volts, set this attribute to 5.0 volts. `setVerticalOffset(myScope, 'Channel1', 5);`
`setVerticalRange`	Specifies the absolute value of the input range the oscilloscope can acquire for the selected channel name as a MATLAB character vector. The units are volts. `setVerticalRange(myScope, 'Channel1', 10);`

Arguments

The Quick-Control Oscilloscope `oscilloscope` function can use the following properties.

Property	Description
`ChannelNames`	Read-only property that provides available channel names in a cell array.
`ChannelsEnabled`	Read-only property that provides currently enabled channel names in a cell array.
`Status`	Read-only property that indicates the communication status. Valid values are `open` or `closed`.
`Timeout`	Get or set a timeout value. Value cannot be a negative number. Default is 10 seconds.
`AcquisitionTime`	Use to get or set acquisition time value. Used to control the time in seconds that corresponds to the record length. Value must be a positive, finite number.
`AcquisitionStartDelay`	Use to set or get the length of time in seconds from the trigger event to first point in waveform record. If positive, the first point in the waveform occurs after the trigger. If negative, the first point in the waveform occurs before the trigger.
`TriggerMode`	Use to set the triggering behavior. Values are: `'normal'` – the oscilloscope waits until the trigger the user specifies occurs. `'auto'` – the oscilloscope automatically triggers if the configured trigger does not occur within the oscilloscope’s timeout period.
`TriggerSlope`	Use to set or get trigger slope value. Valid values are `falling` or `rising`.
`TriggerLevel`	Specifies the voltage threshold in volts for the trigger control.
`TriggerSource`	Specifies the source the oscilloscope monitors for a trigger. It can be channel name or other values.

`Resource`	Set up before connecting to instrument. Set with value of your instrument’s resource string, for example:
	
	`set(myScope, 'Resource',`
	`    'TCPIP0::a-m6104a-004598::inst0::INSTR');`

`DriverDetectionMode`	Optionally used to set up criteria for connection. Valid values are `auto` or `manual`. Default is `auto`. `auto` means you do not have to set a driver name before connecting to an instrument. If set to `manual`, a driver name must be provided before connecting.
`Driver`	Use only if set `DriverDetectionMode` to `manual`. Then use to give driver name. Only use if driver name cannot be figured out programmatically.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Examples

Create an instance of the scope called `myScope`.

```
myScope = oscilloscope()
```

Discover available resources. A resource string is an identifier to the instrument. You need to set it before connecting to the instrument.

```
availableResources = resources(myScope)
```

If multiple resources are available, use your VISA utility to verify the correct resource and set it.

```
set(myScope, 'Resource', 'TCPIP0::a-m6104a-004598::inst0::INSTR');
```

Connect to the scope.

```
connect(myScope);
```

Automatically configure the scope based on the input signal.

```
autoSetup(myScope);
```

Configure the oscilloscope.

```
% Set the acquisition time to 0.01 second.
set(myScope, 'AcquisitionTime', 0.01);

% Set the acquisition to collect 2000 data points.
set(myScope, 'WaveformLength', 2000);

% Set the trigger mode to normal.
set(myScope, 'TriggerMode', 'normal');

% Set the trigger level to 0.1 volt.
set(myScope, 'TriggerLevel', 0.1);

% Enable channel 1.
enableChannel(myScope, 'Channel1');

% Set the vertical coupling to AC.
setVerticalCoupling (myScope, 'Channel1', 'AC');

% Set the vertical range to 5.0.
setVerticalRange (myScope, 'Channel1', 5.0);
```

Communicate with the instrument. For example, read a waveform.

```
% Acquire the waveform.
waveformArray = readWaveform(myScope);

% Plot the waveform and assign labels for the plot.
```

```
plot(waveformArray);
xlabel('Samples');
ylabel('Voltage');
```

See Also

Topics
"The Quick-Control Interfaces" on page 14-20

Introduced in R2011b

propinfo

Instrument object property information

Syntax

```
out = propinfo(obj)
out = propinfo(obj,'PropertyName')
```

Arguments

`obj`	An instrument object.
`'PropertyName'`	A property name or cell array of property names.
`out`	A structure containing property information.

Description

`out = propinfo(obj)` returns the structure `out` with field names given by the property names for `obj`. Each property name in `out` contains the fields shown below.

Field Name	Description
`Type`	The property data type. Possible values are `any`, `ASCII value`, `callback`, `instrument range value`, `double`, `character vector`, and `struct`.
`Constraint`	The type of constraint on the property value. Possible values are `ASCII value`, `bounded`, `callback`, `instrument range value`, `enum`, and `none`.
`ConstraintValue`	Property value constraint. The constraint can be a range of valid values or a list of valid character vector values.
`DefaultValue`	The property default value.
`ReadOnly`	The condition under which a property is read-only. Possible values are `always`, `never`, `whileOpen`, and `whileRecording`.
`Interface Specific`	If the property is interface-specific, a `1` is returned. If a `0` is returned, the property is supported for all interfaces.

`out = propinfo(obj,'PropertyName')` returns the structure `out` for the property specified by *PropertyName*. The field names of `out` are given in the table shown above. If *PropertyName* is a cell array of property names, a cell array of structures is returned for each property.

Examples

To return all property information for the GPIB object `g`,

```
g = gpib('ni',0,1);
out = propinfo(g);
```

To display all the property information for the `InputBufferSize` property,

```
out.InputBufferSize
ans =
                Type: 'double'
          Constraint: 'none'
     ConstraintValue: ''
        DefaultValue: 512
            ReadOnly: 'whileOpen'
   InterfaceSpecific: 0
```

To display the default value for the `EOSMode` property,

```
out.EOSMode.DefaultValue
ans =
none
```

Tips

You can get help for instrument object properties with the `instrhelp` function.

You can display all instrument object property names and their current values using the `get` function. You can display all configurable properties and their possible values using the `set` function.

When specifying property names, you can do so without regard to case, and you can make use of property name completion. For example, if `g` is a GPIB object, then the following commands are all valid.

```
out = propinfo(g,'EOSMode');
out = propinfo(g,'eosmode');
out = propinfo(g,'EOSM');
```

See Also

`instrhelp` | `set` | `get`

Introduced before R2006a

query

(To be removed) Write text to instrument, and read data from instrument

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
out = query(obj,'cmd')
out = query(obj,'cmd','wformat')
out = query(obj,'cmd','wformat','rformat')
[out,count] = query(...)
[out,count,msg] = query(...)
[out,count,msg,datagramaddress,datagramport] = query(...)
```

Arguments

`obj`	An interface object.
`'cmd'`	String that is written to the instrument.
`'wformat'`	Format for written data.
`'rformat'`	Format for read data.
`out`	Contains data read from the instrument.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.
`datagramaddress`	The datagram address.
`datagramport`	The datagram port.

Description

`out = query(obj,'cmd')` writes the string `cmd` to the instrument connected to `obj`. The data read from the instrument is returned to `out`. By default, the `%s\n` format is used for `cmd`, and the `%c` format is used for the returned data.

`out = query(obj,'cmd','wformat')` writes the string `cmd` using the format specified by *`wformat`*.

`wformat` is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sprintf` file I/O format specifications or a C manual for more information.

`out = query(obj,'cmd','wformat','rformat')` writes the string `cmd` using the format specified by *`wformat`*. The data read from the instrument is returned to `out` using the format specified by *`rformat`*.

rformat is a C language conversion specification. The supported conversion specifications are identical to those supported by *wformat*.

`[out,count] = query(...)` returns the number of values read to `count`.

`[out,count,msg] = query(...)` returns a warning message to `msg` if the read operation did not complete successfully.

`[out,count,msg,datagramaddress,datagramport] = query(...)` returns the remote address and port from which the datagram originated. These values are returned only when using a UDP object.

Examples

This example creates the GPIB object `g`, connects `g` to a Tektronix TDS 210 oscilloscope, writes and reads text data using `query`, and then disconnects `g` from the instrument.

```
g = gpib('ni',0,1);
fopen(g)
idn = query(g,'*IDN?')
idn =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
fclose(g)
```

Tips

Before you can write or read data, `obj` must be connected to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a query operation while `obj` is not connected to the instrument.

`query` operates only in synchronous mode, and blocks the command line until the write and read operations complete execution.

Using `query` is equivalent to using the `fprintf` and `fgets` functions. The rules for completing a write operation are described in the `fprintf` reference pages. The rules for completing a read operation are described in the `fgets` reference pages.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Compatibility Considerations

`serial` object interface will be removed
Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See "Transition Your Code to udpport Interface" on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See "Transition Your Code to VISA-GPIB Interface" on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fopen` | `fprintf` | `fgets` | `sprintf`

Properties

Introduced before R2006a

read

Read data from serial port

Syntax

```
data = read(device,count,datatype)
```

Description

`data = read(device,count,datatype)` reads the number of values specified by `count` in the form specified by `datatype` from the serial port connection `device`. For all numeric `datatype` types, `data` is a row vector of double values. For the text type `datatype` values of `"char"` or `"string"`, `data` is of the specified type. The function suspends MATLAB execution until the specified number of values are read or a timeout occurs.

Examples

Write and Read Data with Serial Port Device

Create a connection to a serial port device. In this example, the serial port at `COM3` is connected to a loopback device.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Write the values `[1,2,3,4,5]` in uint8 format.

```
write(device,1:5,"uint8")
```

Since the port is connected to a loopback device, the data you write to the device is returned to MATLAB. Read all the data.

```
read(device,5,"uint8")
```

```
ans = 1×5

     1     2     3     4     5
```

Input Arguments

`device` — Serial port connection
`serialport` object

Serial port connection, specified as a `serialport` object.

Example: `read(device,20,"uint32")` reads data from the serial port connection `device`.

`count` — Number of values to read
numeric

Number of values to read, specified as a positive integer value. If `count` is greater than the `NumBytesAvailable` property of `device`, the function suspends MATLAB execution and waits until the specified amount of data is read or a timeout occurs.

Example: `read(device,5,"uint32")` reads five values of uint32 data. Each uint32 value is four bytes, for a total of 20 bytes read.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`datatype` — Size and format of each value
`"uint8"` | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to read for each value and the interpretation of those bytes as a MATLAB data type.

Example: `read(device,5,"int16")` reads five values of uint16 data. Each uint16 value is two bytes, for a total of 10 bytes read.

Data Types: `char` | `string`

See Also

Functions
`readbinblock` | `readline` | `serialport`

Introduced in R2019b

read

Read data from remote host over TCP/IP

Syntax

```
data = read(t)
data = read(t,count)
data = read(t,count,datatype)
```

Description

`data = read(t)` reads all available numeric or ASCII data from the remote host specified by the TCP/IP client `t` and returns the data as a row or column vector of doubles or text. The number of values read is specified by the `NumBytesAvailable` property of `t`. The function suspends MATLAB execution until the specified number of values are read or a timeout occurs.

`data = read(t,count)` reads `count` number of values and returns the data.

`data = read(t,count,datatype)` reads `count` number of values in the form specified by `datatype` and returns the data. The `datatype` argument is a character vector of a standard MATLAB data type.

Examples

Write and Read uint8 Data from Remote Host

Create a TCP/IP client connection called `t`, connecting to a TCP/IP echo server with port 4000. To do so, you must have an `echotcpip` server running on port 4000.

```
echotcpip("on",4000)
t = tcpclient("localhost",4000)
```

```
t = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

The `write` function synchronously writes data to the remote host connected to `t`. First specify the data and then write the data. The function suspends MATLAB execution until the specified number of values is written to the remote host.

Assign 10 bytes of `uint8` data to the variable `data`.

```
data = uint8(1:10)
```

```
data = 1×10 uint8 row vector

    1    2    3    4    5    6    7    8    9   10
```

View the data.

```
whos data
```

```
  Name      Size            Bytes  Class    Attributes

  data      1x10               10  uint8
```

Write data to the echo server.

```
write(t,data)
```

Confirm the success of the writing operation by viewing the `NumBytesAvailable` property.

```
t.NumBytesAvailable
```

```
ans = 10
```

Since the client is connected to an echo server, the data you write to the server is returned to the client. Read all the bytes of data available.

```
read(t)
```

```
ans = 1×10 uint8 row vector

    1    2    3    4    5    6    7    8    9   10
```

Using the `read` function with no arguments reads all available bytes of data from `t` connected to the remote host and returns the data. The number of values read is determined by the `NumBytesAvailable` property, which is the number of bytes available in the input buffer.

Close the connection between the TCP/IP client and the remote host by clearing the object. Turn off the `echotcpip` server.

```
clear t
echotcpip("off")
```

Specify Size and Data Type to Read from Remote Host

Create a TCP/IP client connection called `t`, connecting to a TCP/IP echo server with port 4000. To do so, you must have an `echotcpip` server running on port 4000.

```
echotcpip("on",4000)
t = tcpclient("localhost",4000)
```

```
t = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
```

```
    NumBytesAvailable: 0

  Show all properties, functions
```

The `write` function synchronously writes data to the remote host connected to `t`. First specify the data and then write the data. The function waits until the specified number of values is written to the remote host.

Assign 10 bytes of data to the variable `data`.

```
data = (1:10)
```

```
data = 1×10

     1     2     3     4     5     6     7     8     9    10
```

View the data.

```
whos data
```

```
  Name      Size            Bytes  Class     Attributes

  data      1x10               80  double
```

Write data to the echo server.

```
write(t,data)
```

Confirm the success of the writing operation by viewing the `NumBytesAvailable` property.

```
t.NumBytesAvailable
```

```
ans = 80
```

For any read or write operation, the data type is converted to `uint8` for the data transfer. After the transfer, the data type reverts to the specified `datatype`. Since one `double` equals eight `uint8` bytes, there are 80 bytes available.

Since the client is connected to an echo server, the data you write to the server is returned to the client. Read 10 doubles from the server. The object name is always the first argument. The `size` argument must be the second argument, and `datatype` must be the third argument.

```
read(t,10,"double")
```

```
ans = 1×10

     1     2     3     4     5     6     7     8     9    10
```

Close the connection between the TCP/IP client and the remote host by clearing the object. Turn off the `echotcpip` server.

```
clear t
echotcpip("off")
```

Input Arguments

t — TCP/IP client
`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `read(t)` reads all available data from the TCP/IP client `t`.

count — Number of values to read
numeric

Number of values to read, specified as a positive integer value. If `count` is greater than the `NumBytesAvailable` property of `t`, the function suspends MATLAB execution and waits until the specified amount of data is read or a timeout occurs.

Example: `read(t,5)` reads five values of `uint8` data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

datatype — Size and format of each value
`"double"` (default) | `"uint8"` | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to read for each value and the interpretation of those bytes as a MATLAB data type.

Example: `read(t,10,"double")` reads 10 values of double data.

Data Types: `char` | `string`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`tcpclient` | `readline` | `write`

Topics
“Create TCP/IP Client and Configure Settings” on page 7-3
“Write and Read Data over TCP/IP Interface” on page 7-7

Introduced in R2014b

read

Read data sent to TCP/IP server

Syntax

```
data = read(t,count)
data = read(t,count,datatype)
```

Description

`data = read(t,count)` reads the number of values specified by `count` sent to the TCP/IP server `t` from the client connected to it and returns the data as a row or column vector of doubles or text. The function suspends MATLAB execution until the specified number of values is read or a timeout occurs.

`data = read(t,count,datatype)` reads the number of values specified by `count` in the form specified by `datatype` and returns the data. The `datatype` argument is a character vector of a standard MATLAB data type. For all numeric `datatype` types, `data` is a row vector of double values. For the text type `datatype` values of `"char"` or `"string"`, `data` is of the specified type.

Examples

Read `uint8` Data Sent to TCP/IP Server

Create a TCP/IP server that listens for a client connection request at the specified port and IP address. Then, read data sent to the server from the connected client.

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:
```

```
            Address: 'localhost'
               Port: 4000
  NumBytesAvailable: 0

Show all properties, functions
```

Display the values of the `Connected`, `ClientAddress`, and `ClientPort` properties for `server`.

```
server
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 1
        ClientAddress: "127.0.0.1"
           ClientPort: 59357
    NumBytesAvailable: 0

  Show all properties, functions
```

The output shows that `server` successfully accepts a request from `client` and that `client` establishes a connection to `server`.

Write data to the TCP/IP client. Since the client is connected to the server, this data is available in the server. Read the data using the `server` object.

```
write(client,[4,8,15,16,23,42],"uint8")
read(server,server.NumBytesAvailable)
```

```
ans = 1×6

     4     8    15    16    23    42
```

Read String Data Sent to TCP/IP Server

Create a TCP/IP server that listens for a client connection request at the specified port and IP address. Then read data sent to the server from the connected client.

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
```

```
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Display the values of the `Connected`, `ClientAddress`, and `ClientPort` properties for `server`.

```
server
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 1
        ClientAddress: "127.0.0.1"
           ClientPort: 65440
    NumBytesAvailable: 0

  Show all properties, functions
```

The output shows that `server` successfully accepts a request from `client` and that `client` establishes a connection to `server`.

Write data to the TCP/IP client. Since the client is connected to the server, this data is available in the server. Read the first five values of string data using the `server` object.

```
write(client,"helloworld","string")
read(server,5,"string")
```

```
ans = 
"hello"
```

If you read five more values, you receive the remaining string data.

```
read(server,5,"string")
```

```
ans = 
"world"
```

Input Arguments

t — TCP/IP server
tcpserver object

TCP/IP server, specified as a tcpserver object.

Example: read(t,5) reads data sent to the TCP/IP server t from the client connected to it.

count — Number of values to read
numeric

Number of values to read, specified as a positive integer value. If count is greater than the NumBytesAvailable property of t, the function suspends MATLAB execution and waits until it reads the specified amount of data or a timeout occurs.

Example: read(device,2) reads two values of uint8 data.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

datatype — Size and format of each value
"uint8" (default) | "int8" | "uint16" | "int16" | "uint32" | "int32" | "uint64" | "int64" | "single" | "double" | "char" | "string"

Size and format of each value, specified as a character vector or string. datatype determines the number of bytes to read for each value and the interpretation of those bytes as a MATLAB data type.

Example: read(t,1,"uint16") reads one value of uint16 data. Each uint16 value is two bytes.

Data Types: char | string

See Also
tcpserver | readline | write

Introduced in R2021a

read

Read data from UDP socket

Syntax

```
data = read(u,count)
data = read(u,count,datatype)
```

Description

`data = read(u,count)` reads the specified number of values from the `udpport` socket `u`, using the default precision of `uint8` to interpret numeric data. For a byte-type `udpport` object `u`, the result `data` is a row vector of doubles. For a datagram-type `udpport` object `u`, the result `data` is a `Datagram` structure or array of structures.

`data = read(u,count,datatype)` reads from the `udpport` socket `u`, with the precision specified by `datatype`. For a numeric `datatype`, the values are returned as double. For `datatype` of `"char"` or `"string"`, the results are the specified type.

Examples

Read Byte Data from UDP Socket

This example shows how to read byte data.

Read 5 values of `uint32` data from the `udpport` socket.

```
u = udpport("IPV4");
data = read(u,5,"uint32");
```

The 5 values comprise a total of 20 bytes of `uint32` data at the `udpport` socket. In MATLAB, `data` is an array of doubles.

Read Datagram Data from UDP Socket

This example shows how to read datagram data.

Turn `echoudp` on at port 3030, then create a datagram-type `udpport` object with an `OutputDatagramSize` of 5.

```
echoudp("on",3030);
u = udpport("datagram","OutputDatagramSize",5);
```

Send 20 bytes of `uint8` data to the `echoudp` port.

```
write(u,1:20,"uint8","127.0.0.1",3030);
```

Because `OutputDatagramSize` is set to 5, the 20 bytes are sent as 4 datagram packets, each containing 5 bytes of data.

Verify that 4 datagrams were received from the echo server.

```
u.NumDatagramsAvailable
```

```
ans =

     4
```

Read the 4 datagrams received from the echo server.

```
data = read(u,u.NumDatagramsAvailable,"uint8")
```

```
data = 

  1×4 Datagram array with properties:

    Data
    SenderAddress
    SenderPort
```

The first datagram contains the values 1-5 (5 bytes), the second 6-10, the third 11-15, and the fourth 16-20.

View the third datagram.

```
data(3)
```

```
ans = 

  Datagram with properties:

             Data: [11 12 13 14 15]
    SenderAddress: "127.0.0.1"
       SenderPort: 3030
```

Input Arguments

u — UDP socket
`udpport` object

UDP socket, specified as a `udpport` object.

Example: `u = udpport`

Data Types: `udpport object`

`count` — Number of values or datagrams to read from `udpport` socket
numeric

Number of values or datagrams to read from `udpport` socket, specified as a numeric value. You cannot specify `count` as `0`, `Inf`, or `NaN`. If `count` is greater than the `NumBytesAvailable` or `NumDatagramsAvailable` property of the `udpport` object, then the function waits until the specified number of values or datagrams are read or until timeout occurs.

Example: `16`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`datatype` — MATLAB data type for each value

`"uint8"` (default) | string | character vector

MATLAB data type for each value, specified as a string or character vector. `datatype` specifies the number of bits to read for each value, and the interpretation of those bits as a MATLAB data type. Allowed values are `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"`, `"double"`, `"single"`, `"char"`, and `"string"`.

Example: `"uint16"`

Data Types: `char` | `string`

Output Arguments

`data` — Values read from `udpport` socket

numeric vector | string | character vector | datagram vector

Values read from the `udpport` socket. For a byte-type `udpport` object, the result is a string, character vector or 1-by-`N` row vector of doubles where `N` is the number of values specified by `count`. For a datagram-type `udpport` object, the result is a `Datagram` structure or array of structures. If no data is returned, the `data` is empty.

See Also

Functions

`udpport` | `readline` | `write` | `writeline`

Introduced in R2020b

read

Read data from VISA resource

Syntax

```
data = read(v,count)
data = read(v,count,datatype)
```

Description

`data = read(v,count)` reads the number of values specified by `count` from the VISA resource `v` and returns the data as a row or column vector of doubles or text. The function suspends MATLAB execution until the specified number of values is read or a timeout occurs.

`data = read(v,count,datatype)` reads the number of values specified by `count` in the form specified by `datatype` and returns the data. The `datatype` argument is a character vector of a standard MATLAB data type. For all numeric `datatype` types, `data` is a row vector of double values. For the text type `datatype` values of `"char"` or `"string"`, `data` is of the specified type.

Examples

Read `uint32` Data from VISA Resource

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Read five values of `uint32` data from the VISA resource `v`.

```
data = read(v,5,"uint32");
```

The five values are a total of 20 bytes of `uint32` data.

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `read(v,5)` reads data from the VISA resource `v`.

`count` — Number of values to read
numeric

Number of values to read, specified as a positive integer value. If `count` is greater than the `NumBytesAvailable` property of `v`, the function suspends MATLAB execution and waits until it reads the specified amount of data or a timeout occurs.

Example: `read(v,2)` reads two values of `uint8` data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`datatype` — Size and format of each value
`"uint8"` (default) | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to read for each value and the interpretation of those bytes as a MATLAB data type.

Example: `read(v,1,"uint16")` reads one value of `uint16` data. Each uint16 value is two bytes.

Data Types: `char` | `string`

See Also
`visadev` | `readline` | `write`

Introduced in R2021a

read

Read binary data from SPI instrument

Syntax

```
A = read(OBJ, SIZE)
```

Description

`A = read(OBJ, SIZE)` reads the specified number of values, SIZE, from the SPI device connected to interface object, OBJ, and returns to A. OBJ must be a 1-by-1 SPI interface object. By default the `'uint8'` precision is used.

The interface object must be connected to the device with the `connect` function before any data can be read from the device, otherwise an error is returned. A connected interface object has a `ConnectionStatus` property value of `connected`.

Available options for SIZE include: `N` – read at most `N` values into a column vector. SIZE cannot be set to `INF`.

The SPI protocol operates in full duplex mode, input and output data transfers happen simultaneously. SPI communication requires N bytes of dummy data to be written into the device for reading N bytes of data from the device. The dummy data written is zeros.

For more information on using the SPI interface and this function, see "Configuring SPI Communication" on page 10-3 and "Transmitting Data Over the SPI Interface" on page 10-7.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Examples

This example shows how to create a SPI object `s`, and read data.

Construct a `spi` object called `s` using `Vendor 'aardvark'`, with `BoardIndex` of `0`, and `Port` of `0`.

```
s = spi('aardvark', 0, 0);
```

Connect to the chip.

```
connect(s);
```

Read data from the chip.

```
data = read(s, 2);
```

Disconnect the SPI device and clean up by clearing the object.

```
disconnect(s);
clear('s');
```

Introduced in R2013b

read

Read data from a MODBUS server

Syntax

```
read(m,target,address)
read(m,target,address,count)
read(m,target,address,count,serverId,precision)
```

Description

`read(m,target,address)` reads one data value to MODBUS object `m` from target type `target` at the starting address `address`. The function reads one value by default. If you want to read more than one value, add the `count` argument.

`read(m,target,address,count)` reads data to MODBUS object `m` from target type `target`at the starting address `address` using the number of values to read `count`.

`read(m,target,address,count,serverId,precision)` additionally specifies `serverId`, which is the address of the server to send the read command to, and the `precision`, which is the data format of the register being read.

Examples

Read Coils Over MODBUS

If the read target is coils, the function reads the values from 1–2000 contiguous coils in the remote server, starting at the specified address. A coil is a single output bit. A value of `1` indicates the coil is on and a value of `0` means it is off.

Read 8 coils, starting at address 1. The `address` parameter is the starting address of the coils to read, and the `count` parameter is the number of coils to read.

```
read(m,'coils',1,8)
```

```
ans =

   1   1   0   1   1   0   1   0
```

Read Inputs Over MODBUS

If the read target is inputs, the function reads the values from 1–2000 contiguous discrete inputs in the remote server, starting at the specified address. A discrete input is a single input bit. A value of `1` indicates the input is on and a value of `0` means it is off.

Read 10 discrete inputs, starting at address 2. The `address` parameter is the starting address of the inputs to read, and the `count` parameter is the number of inputs to read.

```
read(m,'inputs',2,10)

ans =

   1   1   0   1   1   0   1   0   0   1
```

Read Input Registers Over MODBUS

If the read target is input registers, the function reads the values from 1–125 contiguous input registers in the remote server, starting at the specified address. An input register is a 16-bit read-only register.

Read 4 input registers, starting at address 20. The `address` parameter is the starting address of the input registers to read, and the `count` parameter is the number of input registers to read.

```
read(m,'inputregs',20,4)

ans =

   27640   60013   51918   62881
```

Read Holding Registers Over MODBUS

If the read target is holding registers, the function reads the values from 1–125 contiguous holding registers in the remote server, starting at the specified address. A holding register is a 16-bit read/write register.

Read 5 holding registers, starting at address 2. The `address` parameter is the starting address of the holding registers to read, and the `count` parameter is the number of holding registers to read.

```
read(m,'holdingregs',2,5)

ans =

   27640   60013   51918   62881   34836
```

Specify Server ID and Precision Options for the Read Operation

You can read any of the four types of targets and also specify the optional parameters for server ID, and you can specify precision for the two types of registers. You can set either option by itself or set both the `serverId` option and the `precision` option together. Both options should be listed after the required arguments.

Read 8 holding registers starting at address 1 using a precision of `'uint32'` from Server ID 3.

```
read(m,'holdingregs',1,8,3,'uint32');
```

Read Mixed Data Types

You can read contiguous values of different data types (precisions) by specifying the data type for each value. You can do that within the syntax of the `read` function, or set up variables containing arrays of counts and precisions. Both methods are shown here.

Read contiguous registers of the same data type.

```
read(m,'holdingregs',500,10,'uint32');
```

In that example, the target type is holding registers, the starting address is `500`, the count is `10`, and the precision is `uint32`.

If you wanted to have the 10 values be of mixed data types, you can use this syntax:

```
read(m,'holdingregs',500,[3 2 3 2],{'uint16', 'single', 'double', 'int16'});
```

You use an array of values within the command for both count and precision. In this case, the counts are 3, 2, 3, and 2. The command will read 3 values of data type `uint16`, 2 values of data type `single`, 3 values of data type `double`, and 2 values of data type `int16`. The registers are contiguous, starting at address `500`.

Instead of using arrays inside the read command as shown in the previous step, you can also use arrays as variables in the command. The equivalent code for the same example is:

```
count = [3 2 3 2]
precision = {'uint16', 'single', 'double', 'int16'}
read(m,'holdingregs',500,count,precision);
```

Using variables is convenient when you have a lot of values to read and they are of mixed data types.

Input Arguments

`target` — Target area to read
character vector | string

Target area to read, specified as a character vector or string. You can perform a MODBUS read operation on four types of targets: coils, inputs, input registers, and holding registers, corresponding to the values `'coils'`, `'inputs'`, `'inputregs'`, and `'holdingregs'`. Target must be the first argument after the object name. This example reads 8 coils starting at address 1.

Example: `read(m,'coils',1,8)`

Data Types: `char`

`address` — Starting address to read from
double

Starting address to read from, specified as a double. Address must be the second argument after the object name. This example reads 10 coils starting at address 2.

Example: `read(m,'coils',2,10)`

Data Types: `double`

`count` — Number of values to read
double

Number of values to read, specified as a double. Count must be the third argument after the object name. If you do not specify a count, the default of `1` is used. This example reads 12 coils starting at address 2.

Example: `read(m,'coils',2,12)`

Data Types: `double`

`serverId` — Address of the server to send the read command to
double

Address of the server to send the read command to, specified as a double. Server ID must be specified after the object name, target, address, and count. If you do not specify a `serverId`, the default of `1` is used. Valid values are `0-247`, with `0` being the broadcast address.

Note What some devices refer to as a `slaveID` property, may work as a `serverID` property in the MODBUS interface. For some manufacturers a slave ID is sometimes referred to as a server ID. If your device uses a `slaveID` property, it might work to use it as the `serverID` property with the read command as described here.

This example reads 8 coils starting at address 1 from server ID 3.

Example: `read(m,'coils',1,8,3);`

Data Types: `double`

`precision` — Data format of the register being read from on the MODBUS server
character vector | string

Data format of the register being read from on the MODBUS server, specified as a character vector or string. Precision must be specified after the object name, target, address, and count. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

Note that `precision` does not refer to the return type, which is always `'double'`. It specifies how to interpret the register data.

This example reads 6 holding registers starting at address 2 using a precision of `'uint32'`.

Example: `read(m,'holdingregs',2,6,'uint32');`

Data Types: `char`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`modbus` | `write` | `writeRead` | `maskWrite`

Topics

“Create a MODBUS Connection” on page 11-3
“Configure Properties for MODBUS Communication” on page 11-5

“Read Data from a MODBUS Server” on page 11-8
“Read Temperature from a Remote Temperature Sensor” on page 11-13

Introduced in R2017a

readasync

(To be removed) Read data asynchronously from instrument

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
readasync(obj)
readasync(obj,size)
```

Arguments

`obj`	An interface object.
`size`	The number of bytes to read from the instrument.

Description

`readasync(obj)` initiates an asynchronous read operation.

`readasync(obj,size)` asynchronously reads, at most, the number of bytes specified by `size`. If `size` is greater than the difference between the `InputBufferSize` property value and the `BytesAvailable` property value, an error is returned.

Examples

This example creates the serial port object `s`, connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` on a Windows machine to read data asynchronously only if `readasync` is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
fprintf(s,'Measurement:Meas1:Type Pk2Pk')
fprintf(s,'Measurement:Meas1:Value?')
```

Initially, there is no data in the input buffer.

```
s.BytesAvailable
ans =
     0
```

Begin reading data asynchronously from the instrument using `readasync`. When the read operation is complete, return the data to the MATLAB workspace using `fscanf`.

```
readasync(s)
s.BytesAvailable
```

```
ans =
     15
out = fscanf(s)
out =
2.0399999619E0
fclose(s)
```

Tips

Before you can read data, you must connect `obj` to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

For serial port, TCPIP, UDP, and VISA-serial objects, you should use `readasync` only when you configure the `ReadAsyncMode` property to `manual`. `readasync` is ignored if used when `ReadAsyncMode` is `continuous`.

The `TransferStatus` property indicates if an asynchronous read or write operation is in progress. For all interface objects, you cannot use `readasync` while a read operation is in progress. For serial port and VISA-serial objects, you can write data while an asynchronous read is in progress because serial ports have separate read and write pins. You can stop asynchronous read and write operations with the `stopasync` function.

You can monitor the amount of data stored in the input buffer with the `BytesAvailable` property. Additionally, you can use the `BytesAvailableFcn` property to execute a callback function when the terminator or the specified amount of data is read.

Asynchronous operation is not supported for NI VISA objects on the UNIX platform. So if you use the `readasync` function with a NI VISA object, you will get an error.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Rules for Completing an Asynchronous Read Operation

An asynchronous read operation with `readasync` completes when one of these conditions is met:

- The terminator is read. For serial port, TCPIP, UDP, and VISA-serial objects, the terminator is given by the `Terminator` property. Note that for UDP objects, `DatagramTerminateMode` must be `off`.

  For all other interface objects except VISA-RSIB, the terminator is given by the `EOSCharCode` property.
- The time specified by the `Timeout` property passes.
- The specified number of bytes is read.
- The input buffer is filled.
- A datagram has been received (UDP objects only if `DatagramTerminateMode` is `on`)
- The EOI line is asserted (GPIB and VXI instruments only).

For serial port, TCPIP, UDP, and VISA-serial objects, `readasync` can be slow because it checks for the terminator. To increase speed, you might want to configure `ReadAsyncMode` to `continuous` and continuously return data to the input buffer as soon as it is available from the instrument.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See "Transition Your Code to serialport Interface" on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See "Transition Your Code to bluetooth Interface" for more information about using the recommended functionality.

`tcpip` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See "Transition Your Code to tcpclient Interface" on page 7-36 or "Transition Your Code to tcpserver Interface" on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed

Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See "Transition Your Code to udpport Interface" on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed

Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

gpib object interface will be removed

Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fopen` | `stopasync`

Properties
BytesAvailable | BytesAvailableFcn | ReadAsyncMode | Status | TransferStatus

Introduced before R2006a

readbinblock

Read one binblock of data from serial port

Syntax

```
data = readbinblock(device)
data = readbinblock(device,precision)
```

Description

`data = readbinblock(device)` reads a binblock of data from the serial port as uint8 numeric values and returns a 1-by-N array of doubles.

`data = readbinblock(device,precision)` reads a binblock of data interpreted as the type specified by `precision`. For numeric types, the data is returned as a 1-by-N array of doubles. For text types, the data is returned as character vector or string, as specified.

The function blocks MATLAB and waits until a binblock is read from the serial port.

Examples

Read Binblock of uint8 Data

Read a binblock of numeric uint8 data from the serial port.

The default precision is uint8.

```
s = serialport("COM3",9600);
data = readbinblock(s);
```

Read Binblock of uint16 Data

Read a binblock of numeric uint16 data from the serial port.

```
s = serialport("COM3",9600);
data = readbinblock(s,"uint16")
```

Input Arguments

device — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

`precision` — Size and format of each value
`'uint8'` (default) | `'int8'` | `'uint16'` | `'int16'` | `'uint32'` | `'int32'` | `'uint64'` | `'int64'` | `'single'` | `'double'` | `'char'` | `'string'`

Size and format of binblock data, specified as a character vector or string. `precision` determines the number of bits to read for each value and the interpretation of those bits as a MATLAB data type.

Example: `'int16'`

Data Types: `char` | `string`

Output Arguments

`data` — Numeric or ASCII data
1-by-N double | string | char

Numeric or ASCII data, returned as a 1-by-N vector of doubles or text. For all numeric `precision` types, `data` is a row vector of double values. For the text type `precision` values of `'char'` or `'string'`, `data` is of the specified type.

See Also

Functions
`read` | `readline` | `serialport` | `serialportlist`

Introduced in R2019b

readbinblock

Read one binblock of data from remote host over TCP/IP

Syntax

```
data = readbinblock(t)
data = readbinblock(t,datatype)
```

Description

`data = readbinblock(t)` reads a binblock of data from the remote host specified by the TCP/IP client `t` and returns the data as a row vector of doubles. The function suspends MATLAB execution until the specified number of values are read or a timeout occurs.

`data = readbinblock(t,datatype)` reads a binblock of data interpreted as the type specified by `datatype`. For numeric types, the data is returned as a row vector of doubles. For text types, the data is returned as a character vector or string, as specified.

Examples

Write and Read Binblock of Data from Remote Host

Create a TCP/IP client connection called `t`, connecting to a TCP/IP echo server with port 4000. To do so, you must have an `echotcpip` server running on port 4000.

```
echotcpip("on",4000)
t = tcpclient("localhost",4000)
```

```
t =
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Write the values `[1,2,3,4,5]` as a binblock in uint8 format.

```
writebinblock(t,1:5,"uint8")
```

Write another binblock of data. Write the values `[6,7,8,9,10]` as double data.

```
writebinblock(t,6:10,"double")
```

Since the client is connected to an echo server, the data you write to the server is returned to the client. Read the first binblock of data that you wrote.

```
readbinblock(t)
```

```
ans = 1×5

     1     2     3     4     5
```

Read a binblock of data again to return the second set of values that you wrote. Specify the data as `double`.

```
readbinblock(t,"double")
```

```
ans = 1×5

     6     7     8     9    10
```

Close the connection between the TCP/IP client and the remote host by clearing the object. Turn off the `echotcpip` server.

```
clear t
echotcpip("off")
```

Input Arguments

t — TCP/IP client
`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `readbinblock(t)` reads a binblock of data from the TCP/IP client `t`.

datatype — Size and format of each value
`"uint8"` (default) | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to read for each value and the interpretation of those bytes as a MATLAB data type.

Example: `readbinblock(t,"double")` reads a binblock of double data.

Data Types: `char` | `string`

See Also

`tcpclient` | `read` | `readline` | `writebinblock`

Introduced in R2020b

readbinblock

Read one binblock of data sent to TCP/IP server

Syntax

```
data = readbinblock(t)
data = readbinblock(t,datatype)
```

Description

`data = readbinblock(t)` reads a binblock of data sent to the TCP/IP server `t` from the client connected to it and returns the data as a row vector of doubles. The function suspends MATLAB execution until the number of values specified in the binblock is read or a timeout occurs.

`data = readbinblock(t,datatype)` reads a binblock of data interpreted as the type specified by `datatype`. For numeric types, the data is returned as a row vector of doubles. For text types, the data is returned as a character vector or string, as specified.

Examples

Read Binblock of Data Sent to TCP/IP Server

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0
```

```
  Show all properties, functions
```

Write a binblock of data to the client in `uint8` format.

```
writebinblock(client,[4,8,15,16,23,42],"uint8")
```

Since the client is connected to the server, the data you write to the client is available to be read from the `server` object. Read the binblock of data.

```
readbinblock(server)
```

```
ans = 1×6

     4     8    15    16    23    42
```

If you write a binblock of data to the client that is not in `uint8` format, you must specify the data type when reading it from the `server` object.

```
writebinblock(client,"Hello, world!","string")
readbinblock(server,"string")
```

```
ans = 
"Hello, world!"
```

Input Arguments

t — TCP/IP server
`tcpserver` object

TCP/IP server, specified as a `tcpserver` object.

Example: `readbinblock(t)` reads a binblock of data sent to the TCP/IP server `t` from the client connected to it.

datatype — Size and format of each value
`"uint8"` (default) | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to read for each value and the interpretation of those bytes as a MATLAB data type.

Example: `readbinblock(t,"double")` reads a binblock of double data.

Data Types: `char` | `string`

See Also

`tcpserver` | `read` | `readline` | `writebinblock`

Introduced in R2021a

readbinblock

Read one binblock of data from VISA resource

Syntax

```
data = readbinblock(v)
data = readbinblock(v,datatype)
```

Description

`data = readbinblock(v)` reads a binblock of data from the VISA resource `v` and returns the data as a row vector of doubles. The function suspends MATLAB execution until the number of values specified in the binblock is read or a timeout occurs.

`data = readbinblock(v,datatype)` reads a binblock of data interpreted as the type specified by `datatype`. For numeric types, the data is returned as a row vector of doubles. For text types, the data is returned as a character vector or string, as specified.

Examples

Read Binblock of Data from VISA Resource

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Read a binblock of `uint8` data from the VISA resource `v`.

```
data = readbinblock(v);
```

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `readbinblock(v)` reads a binblock of data from the VISA resource `v`.

datatype — Size and format of each value
`"uint8"` (default) | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to read for each value and the interpretation of those bytes as a MATLAB data type.

Example: `readbinblock(v,"double")` reads a binblock of double data.

Data Types: `char` | `string`

See Also

`visadev` | `read` | `readline` | `writebinblock`

Introduced in R2021a

readline

Read line of ASCII string data from serial port

Syntax

```
data = readline(device)
```

Description

`data = readline(device)` reads ASCII data until the first occurrence of the terminator from the serial port connection and returns `data` as a string without the terminator. The function suspends MATLAB execution until the terminator is reached or a timeout occurs.

Examples

Write and Read Line of ASCII Data from Serial Port Device

Create a connection to a serial port device. In this example, the serial port at `COM3` is connected to a loopback device.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Check the default ASCII terminator.

```
device.Terminator
```

```
ans = 

    "LF"
```

Set the terminator to "CR" and write a string of ASCII data. The `writeline` function automatically appends the terminator to the data.

```
configureTerminator(device,"CR")
writeline(device,"hello")
```

Write another string of ASCII data with the terminator automatically appended.

```
writeline(device,"world")
```

Since the port is connected to a loopback device, the data you write to the device is returned to MATLAB. Read a string of ASCII data. The `readline` function returns data until it reaches a terminator.

```
readline(device)
```

```
ans = 

    "hello"
```

Read a string of ASCII data again to return the second string that you wrote.

```
readline(device)
```

```
ans = 

    "world"
```

Clear the serial port connection.

```
clear device
```

Input Arguments

device — Serial port connection
`serialport` object

Serial port connection, specified as a `serialport` object.

Example: `readline(device)` reads ASCII data from the serial port connection `device`.

See Also

Functions
`configureTerminator` | `read` | `readbinblock` | `serialport` | `serialportlist` | `writeread` | `writeline`

Introduced in R2019b

readline

Read line of ASCII string data from remote host over TCP/IP

Syntax

```
data = readline(t)
```

Description

`data = readline(t)` reads ASCII data until the first occurrence of the terminator from the remote host specified by the TCP/IP client `t` and returns `data` as a string without the terminator. The function suspends MATLAB execution until the terminator is reached or a timeout occurs.

Examples

Write and Read Line of ASCII Data from Remote Host

Create a TCP/IP client connection called `t`, connecting to a TCP/IP echo server with port 4000. To do so, you must have an `echotcpip` server running on port 4000.

```
echotcpip("on",4000)
t = tcpclient("localhost",4000)
```

```
t = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Check the default ASCII terminator.

```
t.Terminator
```

```
ans = 
"LF"
```

Set the terminator to `"CR"` and write a string of ASCII data. The `writeline` function automatically appends the terminator to the data.

```
configureTerminator(t,"CR")
writeline(t,"hello")
```

Write another string of ASCII data with the terminator automatically appended.

```
writeline(t,"world")
```

Since the client is connected to an echo server, the data you write to the server is returned to the client. Read a string of ASCII data. The `readline` function returns data until it reaches a terminator.

```
readline(t)
```

```
ans = 
"hello"
```

Read a string of ASCII data again to return the second string that you wrote.

```
readline(t)
```

```
ans = 
"world"
```

Close the echo server and clear the TCP/IP client connection.

```
echotcpip("off")
clear t
```

Input Arguments

t — TCP/IP client
`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `readline(t)` reads ASCII data from the remote host specified by the TCP/IP client `t`.

See Also

Functions
`tcpclient` | `read` | `configureTerminator` | `writeline`

Introduced in R2020b

readline

Read line of ASCII string data sent to TCP/IP server

Syntax

```
data = readline(t)
```

Description

`data = readline(t)` reads ASCII data sent to the TCP/IP server `t` from the client connected to it until the first occurrence of the terminator and returns `data` as a string without the terminator. The function suspends MATLAB execution until it reaches a terminator or a timeout occurs.

Examples

Read Line of ASCII Data Sent to TCP/IP Server

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Check the default ASCII terminator for the server.

```
server.Terminator
```

```
ans = 
"LF"
```

Set the terminators for both the server and client to "CR". The TCP/IP server and its connected client must have the same terminator.

```
configureTerminator(server,"CR")
configureTerminator(client,"CR")
```

Write a string of ASCII data to the client. The `writeline` function automatically appends the terminator to the data.

```
writeline(client,"First message.")
```

Write another string of ASCII data with the terminator automatically appended.

```
writeline(client,"Second message.")
```

Since the client is connected to the server, the data you send to the client is available to be read from the server. Read a string of ASCII data received by the server from the client. The `readline` function returns data until it reaches a terminator.

```
readline(server)
```

```
ans = 
"First message."
```

Read a string of ASCII data again to return the second string.

```
readline(server)
```

```
ans = 
"Second message."
```

Input Arguments

t — TCP/IP server

`tcpserver` object

TCP/IP server, specified as a `tcpserver` object.

Example: `readline(t)` reads ASCII data sent to the TCP/IP server `t` from the client connected to it.

See Also

`tcpserver` | `read` | `configureTerminator` | `writeline`

Introduced in R2021a

readline

Read line of ASCII string data from UDP socket

Syntax

```
data = readline(u)
```

Description

`data = readline(u)` reads data from the specified UDP socket until the first occurrence of the terminator, and assigns the result to `data` as a `string`. `u` must be a byte-type `udpport` object. This function waits until the terminator is read or a timeout occurs.

Examples

Read ASCII Line from UDP Socket

Create a UDP socket object and define its terminator.

```
u = udpport;
configureTerminator(u,"CR/LF");
```

Read an ASCII-terminated string from the `udpport` socket.

```
data = readline(u);
```

`data` does not include the terminator.

Input Arguments

u — UDP socket
`udpport` object

UDP socket, specified as a byte-type `udpport` object.

Example: `u = udpport("byte")`

Data Types: `udpport object`

Output Arguments

data — String read from `udpport` socket
string

String value read from the `udpport` socket, returned as a string.

See Also

Functions
`udpport` | `writeline` | `configureTerminator`

Introduced in R2020b

readline

Read line of ASCII string data from VISA resource

Syntax

```
data = readline(v)
```

Description

`data = readline(v)` reads ASCII data from the VISA resource `v` until the first occurrence of the terminator and returns `data` as a string without the terminator. The function suspends MATLAB execution until it reaches a terminator or a timeout occurs.

Examples

Read ASCII Line from VISA Resource

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Set the terminator to `"CR/LF"`.

```
configureTerminator(v,"CR/LF")
```

Read a string of ASCII data from the VISA resource `v`. The `readline` function returns data until it reaches a terminator.

```
data = readline(v);
```

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `readline(v)` reads ASCII data from the VISA resource `v`.

See Also

`visadev` | `read` | `configureTerminator` | `writeline`

Introduced in R2021a

readWaveform

Returns waveform displayed on scope

Syntax

```
w = readWaveform(myScope);
w = readWaveform(myScope, 'acquisition', true);
w = readWaveform(myScope, 'acquisition', false);
```

Description

`w = readWaveform(myScope);` returns waveform(s) displayed on the scope screen. Retrieves the waveform(s) from enabled channel(s). By default it downloads the captured waveform from the scope without acquisition.

`w = readWaveform(myScope, 'acquisition', true);` initiates an acquisition and returns waveform(s) from the oscilloscope.

`w = readWaveform(myScope, 'acquisition', false);` gets waveform from the enabled channel without acquisition

This function can only be used with the `oscilloscope` object. You can use the `getWaveform` function to download the current waveform from the scope or to initiate the waveform and capture it. See the examples below for the three possible use cases.

Note This is the getWaveform function. In R2017a the name changed from getWaveform to readWaveform. The getWaveform function will continue to be supported.

Examples

Use this example if you have captured the waveform(s) using the oscilloscope's front panel and want to download it to the Instrument Control Toolbox for further analysis.

```
o = oscilloscope()
set (o, 'Resource', 'instrumentResourceString');
connect(o);
w = getWaveform(o);
```

Replace `'instrumentResourceString'` with the resource string for your instrument.

Use this example to get the waveform from a circuit output (without configuring the trigger) and download it to the Instrument Control Toolbox to check it.

```
o = oscilloscope()
set (o, 'Resource', 'instrumentResourceString');
connect(o);
enableChannel(o,'Channel1');
w = getWaveform(o);
```

Replace `'instrumentResourceString'` with the resource string for your instrument.

Use this example to capture synchronized input/output signals of a filter circuit when a certain trigger condition is met, stop the acquisition, and download the waveforms to the Instrument Control Toolbox.

```
o = oscilloscope()
set (o, 'Resource', 'instrumentResourceString');
connect(o);
set (o, 'TriggerMode','normal');
enableChannel(o, {'Channel1','Channel2'});
[w1, w2] = readWaveform(o, 'acquisition', true);
```

Replace `'instrumentResourceString'` with the resource string for your instrument.

See Also

Topics
“The Quick-Control Interfaces” on page 14-20

Introduced in R2011b

record

Record data and event information to file

Syntax

```
record(obj)
record(obj,'switch')
```

Arguments

`obj`	An instrument object.
`'switch'`	Switch recording capabilities on or off.

Description

`record(obj)` toggles the recording state for `obj`.

`record(obj,'switch')` initiates or terminates recording for `obj`. *`switch`* can be `on` or `off`. If *`switch`* is `on`, recording is initiated. If *`switch`* is `off`, recording is terminated.

Examples

This example creates the GPIB object `g`, connects `g` to the instrument, and configures `g` to record detailed information to the disk file MyGPIBFile.txt.

```
g = gpib('ni',0,1);
fopen(g)
g.RecordDetail = 'verbose';
g.RecordName = 'MyGPIBFile.txt';
```

Initiate recording, write the `*IDN?` command to the instrument, and read back the identification information.

```
record(g,'on')
fprintf(g,'*IDN?')
out = fscanf(g);
```

Terminate recording and disconnect `g` from the instrument.

```
record(g,'off')
fclose(g)
```

Tips

Before you can record information to disk, `obj` must be connected to the instrument with the `fopen` function. A connected instrument object has a `Status` property value of `open`. An error is returned if you attempt to record information while `obj` is not connected to the instrument. Each instrument object must record information to a separate file. Recording is automatically terminated when `obj` is disconnected from the instrument with `fclose`.

The `RecordName` and `RecordMode` properties are read-only while `obj` is recording, and must be configured before using `record`.

For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to “Debugging: Recording Information to Disk” on page 17-5.

See Also

`fclose` | `fopen` | `propinfo` | RecordMode | RecordName | RecordStatus | Status

Introduced before R2006a

remove

Remove entry from IVI configuration store object

Syntax

```
remove(obj, 'type', 'name')
remove(obj, struct)
```

Arguments

`obj`	IVI configuration store object
`'type'`	Type of entry being removed; *type* can be `DriverSession`, `HardwareAsset`, or `LogicalName`
`'name'`	Name of the `DriverSession`, `HardwareAsset`, or `LogicalName` to be removed
`struct`	Structure defining entries to be removed

Description

`remove(obj, 'type', 'name')` removes an entry of type, *type*, with name, `name`, from the IVI configuration store object, `obj`. *type* can be `HardwareAsset`, `DriverSession`, or `LogicalName`. If an entry of type, *type*, with name, `name`, does not exist, an error will occur.

`remove(obj, struct)` removes an entry using the fields in `struct`. If an entry with the *type* and `name` field in `struct` does not exist, an error will occur.

The modified configuration store object, `obj`, can be saved to the configuration store data file with the `commit` function.

If you attempt to remove an entry that is actively referenced by another entry, an error will occur. For example, you cannot remove a hardware asset that is currently referenced by a driver session.

Examples

```
c = iviconfigurationstore;
remove(c, 'HardwareAsset', 'gpib1');
```

See Also

`iviconfigurationstore` | `add` | `commit` | `update`

Introduced before R2006a

resources

List of available instrument resources for Quick-Control interfaces

Syntax

```
ResourceList = resources(rf)
```

Description

`ResourceList = resources(rf)` lists the resources for RF signal generator object `rf`. It returns a cell array of resources for the Quick-Control RF Signal Generator, Quick-Control Oscilloscope, or Quick-Control Function Generator objects.

Examples

List Resources and Connect to RF Signal Generator

The `resources` function lists resources available for any of the Quick-Control interface objects: RF signal generator (rfsiggen), oscilloscope (oscilloscope), or function generator (fgen). This example uses Quick-Control RF Signal Generator, but the function also works in the same way for the other two object types.

Create an RF signal generator object without assigning the resource or driver.

```
rf = rfsiggen;
```

List the resources.

```
ResourceList = resources(rf)
```

```
ResourceList =

   3x1 cell array

   {'ASRL::COM1'}
   {'ASRL::COM3'}
   'TCPIP0::172.28.22.99::inst0::INSTR'
```

In this case, it finds two COM ports that could host an instrument, and the VISA resource string of an RF signal generator.

Set the RF signal generator resource using the `Resource` property, which is the VISA resource string.

```
rf.Resource = 'TCPIP0::172.28.22.99::inst0::INSTR';
```

Set the RF Signal Generator driver using the `Driver` property, which is a string containing the name of your instrument driver.

```
rf.Driver = 'AgRfSigGen';
```

You can now connect to the instrument.

```
connect(rf);
```

Output Arguments

`ResourceList` — List of instrument resources
cell array of strings

List of instrument resources, returned as a cell array of strings. It represents the VISA resource string for the instrument. The `resources` function can list resources available for any of the Quick-Control interface objects: RF signal generator, oscilloscope, or function generator.

See Also

`rfsiggen` | `start` | `download` | `drivers`

Topics
"Download and Generate Signals with RF Signal Generator" on page 14-45
"Quick-Control RF Signal Generator Functions" on page 14-41
"Quick-Control RF Signal Generator Properties" on page 14-43

Introduced in R2017b

resolvehost

Resolve network host name or IP address

Syntax

```
name = resolvehost(host)
name = resolvehost(host,"name")
address = resolvehost(host,"address")
[name,address] = resolvehost(host)
```

Description

`name = resolvehost(host)` and `name = resolvehost(host,"name")` return the name of the specified `host`, where `host` is the network name or IP address.

`address = resolvehost(host,"address")` returns the address of the specified `host`.

`[name,address] = resolvehost(host)` returns both the name and address of the specified `host`.

Examples

Identify Network Name from Host Address

Provide an IP address to view the associated host name.

```
name = resolvehost("144.212.244.17")
```

```
name =
'www-ahprod.mathworks.com'
```

You can specify the optional input argument `"name"` as well.

```
name = resolvehost("144.212.244.17","name")
```

```
name =
'www-ahprod.mathworks.com'
```

Specifying the host name instead returns the same host name.

```
name = resolvehost("www.mathworks.com")
```

```
name =
'www.mathworks.com'
```

Resolve IP Address from Host Name

Provide a network host name to view the associated IP address. You must include the input argument `"address"`.

```
address = resolvehost("en.wikipedia.org","address")
```

```
address =
'208.80.154.224'
```

Specifying the IP address instead returns the same IP address.

```
address = resolvehost("208.80.154.224","address")
```

```
address =
'208.80.154.224'
```

Resolve Network Name and IP Address from Host

You can view both the network name and IP address of a specified host. You can specify either the name or IP address. In this example, the `resolvehost` function returns the name and address from the provided host name.

```
[name,address] = resolvehost("en.wikipedia.org")
```

```
name =
'en.wikipedia.org'

address =
'208.80.154.224'
```

You can also specify the IP address instead.

```
[name,address] = resolvehost("208.80.154.224")
```

```
name =
'text-lb.eqiad.wikimedia.org'

address =
'208.80.154.224'
```

Input Arguments

`host` — Network host name or address

character vector | string

Network host name or IP address, specified as a character vector or string scalar.

Data Types: `char` | `string`

Output Arguments

`name` — Network name

character vector

Network name, returned as a character vector.

`address` — IP address

character vector

IP address, returned as a character vector.

See Also

`tcpclient` | `udpport`

Introduced before R2006a

rfsiggen

Create Quick-Control RF Signal Generator object

Syntax

```
rf = rfsiggen()
rf = rfsiggen(Resource)
rf = rfsiggen(Resource, Driver)
```

Description

`rf = rfsiggen()` creates the RF signal generator object `rf` to communicate with an RF signal generator instrument. You must specify a resource or a resource and driver later.

`rf = rfsiggen(Resource)` additionally specifies the `Resource`, and connects it to the RF signal generator instrument designated by the resource. This is the VISA resource string for the instrument.

`rf = rfsiggen(Resource, Driver)` additionally specifies the `Driver`, which is the underlying driver to use with the instrument. If it is not specified, the driver is auto-detected.

Examples

Create an RF Signal Generator Object

You can create the `rfsiggen` object without setting the resource, and then set it after object creation.

Create the RF Signal Generator object with no arguments.

```
rf = rfsiggen()
```

Find available resources using the `resources` function.

```
targets = resources(rf)
```

It returns a list of possible VISA resource strings, for example `TCPIP0::172.28.22.99::inst0::INSTR`.

Set the RF signal generator resource using the `Resource` property.

```
rf.Resource = 'TCPIP0::172.28.22.99::inst0::INSTR';
```

Connect to the instrument. When you assign the resource after object creation, you need to explicitly connect to the instrument.

```
connect(rf);
```

Create an RF Signal Generator Object and Set Resource and Driver

You can create the `rfsiggen` object and set the resource and driver during object creation. If those properties are valid, the object automatically connects to the instrument.

Create the RF signal generator object and connect using the specified resource string and driver.

```
rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRfSigGen')
```

Input Arguments

`Resource` — VISA resource string for your instrument
string

VISA resource string for your instrument, specified as a string. Set this before connecting to the instrument. Setting it during object creation is optional and can be used if you know the resource string for your instrument. Otherwise, you can set it after object creation.

Example: `rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR')`

Data Types: `char` | `string`

`Driver` — Underlying driver to use with the instrument
string

Underlying driver to use with the instrument, specified as a string. Set this before connecting to the instrument. If it is not specified, the driver is auto-detected.

Example: `rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRFSigGen')`

Data Types: `char` | `string`

See Also
`download` | `start` | `resources` | `drivers`

Topics
“Download and Generate Signals with RF Signal Generator” on page 14-45
“Quick-Control RF Signal Generator Functions” on page 14-41
“Quick-Control RF Signal Generator Properties” on page 14-43

Introduced in R2017b

save

Save instrument objects and variables to MAT-file

Syntax

```
save filename
save filename obj1 obj2 ...
```

Arguments

`filename`	The MAT-file name.
`obj1 obj2 ...`	Instrument objects or arrays of instrument objects.

Description

`save filename` saves all MATLAB variables to the MAT-file `filename`. If an extension is not specified for `filename`, then a `.mat` extension is used.

`save filename obj1 obj2 ...` saves the instrument objects `obj1 obj2` ... to the MAT-file `filename`.

Examples

This example illustrates how to use the command form and the functional form of `save`.

```
s = serial('COM1');
set(s,'BaudRate',2400,'StopBits',1)
save MySerial1 s
set(s,'BytesAvailableFcn',@mycallback)
save('MySerial2','s')
```

Tips

You can use `save` in the functional form as well as the command form shown above. When using the functional form, you must specify the filename and instrument objects as character vectors. For example, on a Windows machine, save the serial port object `s` to the file `MySerial.mat`,

```
s = serial('COM1');
save('MySerial','s')
```

Any data that is associated with the instrument object is not automatically stored in the MAT-file. For example, suppose there is data in the input buffer for `obj`. To save that data to a MAT-file, you must bring the data into the MATLAB workspace using one of the synchronous read functions, and then save the data to the MAT-file using a separate variable name. You can also save data to a text file with the `record` function.

You return objects and variables to the MATLAB workspace with the `load` command. Values for read-only properties are restored to their default values upon loading. For example, the `Status` property

is restored to `closed`. To determine if a property is read-only, examine its reference pages or use the `propinfo` function.

See Also

`instrhelp` | `load` | `propinfo` | `record` | Status

Introduced before R2006a

scanstr

(To be removed) Read data from instrument, format as text, and parse

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
A = scanstr(obj)
A = scanstr(obj,'delimiter')
A = scanstr(obj,'delimiter','format')
[A,count] = scanstr(...)
[A,count,msg] = scanstr(...)
```

Arguments

`obj`	An interface object.
`'delimiter'`	One or more delimiters used to parse the data.
`'format'`	C language conversion specification.
`A`	Data read from the instrument and formatted as text.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.

Description

`A = scanstr(obj)` reads formatted data from the instrument connected to `obj`, parses the data using both a comma and a semicolon delimiter, and returns the data to the cell array `A`. Each element of the cell array is determined to be either a double or a character vector.

`A = scanstr(obj,'delimiter')` parses the data into separate variables based on the specified `delimiter`. `delimiter` can be a single character or a character vector array. If `delimiter` is a character vector array, then each character in the array is used as a delimiter.

`A = scanstr(obj,'delimiter','format')` converts the data according to the specified *`format`*. `A` can be a matrix or a cell array depending on *`format`*. See the `textread` help for complete details. *`format`* is a string containing C language conversion specifications.

Conversion specifications involve the % character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. See the `sscanf` file I/O format specifications or a C manual for complete details.

If *`format`* is not specified, then the best format (either a double or a character vector) is chosen.

`[A,count] = scanstr(...)` returns the number of values read to `count`.

`[A,count,msg] = scanstr(...)` returns a warning message to `msg` if the read operation did not complete successfully.

Examples

Create the GPIB object `g` associated with a National Instruments board with index 0 and primary address 2, and connect `g` to a Tektronix TDS 210 oscilloscope.

```
g = gpib('ni',0,2);
fopen(g)
```

Return identification information to separate elements of a cell array using the default delimiters.

```
fprintf(g,'*IDN?');
idn = scanstr(g)
idn =
    'TEKTRONIX'
    'TDS 210'
    [        0]
    'CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04'
```

Tips

Before you can read data from the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read — including the terminator — each time `scanstr` is issued.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed

Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See “Transition Your Code to bluetooth Interface” for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See “Transition Your Code to tcpclient Interface” on page 7-36 or “Transition Your Code to tcpserver Interface” on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See “Transition Your Code to udpport Interface” on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fopen` | `fscanf` | `instrhelp` | `sscanf` | `textread`

Properties
EOSCharCode | EOSMode | Status | Terminator | ValuesReceived

Introduced before R2006a

selftest

Run instrument self-test

Syntax

```
out = selftest(obj)
```

Arguments

`obj`	A device object.
`out`	The result of the self-test.

Description

`out = selftest(obj)` runs the self-test for the instrument associated with the device object specified by `obj`. The result of the self-test is returned to `out`. Note that the test result will vary based on the instrument.

Introduced before R2006a

serial

(To be removed) Create serial port object

Note `serial` will be removed in a future release. Use `serialport` instead. For more information, see "Compatibility Considerations"

Syntax

```
obj = serial('port')
obj = serial('port','PropertyName',PropertyValue,...)
```

Arguments

`'port'`	The serial port name.
`'PropertyName'`	A serial port property name.
`PropertyValue`	A property value supported by *PropertyName*.
`obj`	The serial port object.

Description

`obj = serial('port')` creates a serial port object associated with the serial port specified by `port`. If `port` does not exist, or if it is in use, you will not be able to connect the serial port object to the instrument with the `fopen` function.

`obj = serial('port','PropertyName',PropertyValue,...)` creates a serial port object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and the serial port object is not created.

Examples

This example creates the serial port object `s1` on a Windows machine associated with the serial port COM1.

```
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```
s1.Type
ans =
    serial

s1.Name
ans =
    Serial-COM1

s1.Port
ans =
    COM
```

To specify properties during object creation,

```
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

Tips

At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with serial port objects.

```
instrhelp serial
```

When you create a serial port object, these property values are automatically configured:

- `Type` is given by `serial`.
- `Name` is given by concatenating `Serial` with the port specified in the `serial` function.
- `Port` is given by the port specified in the `serial` function.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, the following commands are all valid.

```
s = serial('COM1','BaudRate',4800);
s = serial('COM1','baudrate',4800);
s = serial('COM1','BAUD',4800);
```

Before you can communicate with the instrument, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while `obj` is not connected to the instrument. You can connect only one serial port object to a given serial port.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Compatibility Considerations

`serial` function will be removed

Not recommended starting in R2019b

`serial` and its object properties will be removed in a future release. Use `serialport` and its properties instead.

This example shows how to connect to a serial port device using the recommended functionality.

Functionality	Use This Instead

`s = serial("COM1");`
`s.BaudRate = 115200;`
`fopen(s)`
	`s = serialport("COM1",115200);`

See “Transition Your Code to serialport Interface” on page 6-26 for more information about using the recommended functionality.

See Also

Functions
`serialport`

Introduced before R2006a

serialbreak

(To be removed) Send break to instrument

Note This `serial` object function will be removed in a future release. Use `serialport` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
serialbreak(obj)
serialbreak(obj,time)
```

Arguments

`obj`	A serial port object.
`time`	The duration of the break, in milliseconds.

Description

`serialbreak(obj)` sends a break of 10 milliseconds to the instrument connected to `obj`.

`serialbreak(obj,time)` sends a break to the instrument with a duration, in milliseconds, specified by `time`. Note that the duration of the break might be inaccurate under some operating systems.

Tips

For some instruments, the break signal provides a way to clear the hardware buffer.

Before you can send a break to the instrument, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to send a break while `obj` is not connected to the instrument.

`serialbreak` is a synchronous function, and blocks the command line until execution is complete.

If you issue `serialbreak` while data is being asynchronously written, an error is returned. In this case, you must call the `stopasync` function or wait for the write operation to complete.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” on page 6-26 for more information about using the recommended functionality.

See Also

`fopen` | `stopasync` | Status

Introduced before R2006a

serialport

Connection to serial port

Description

A `serialport` object represents a serial client for communication with the serial port. After creating the object, use dot notation to set its properties.

Creation

Syntax

```
s = serialport(port,baudrate)
s = serialport(port,baudrate,Name,Value)
s = serialport
```

Description

`s = serialport(port,baudrate)` connects to the serial port specified by `port` with a baud rate of `baudrate`.

`s = serialport(port,baudrate,Name,Value)` connects to the serial port and sets additional properties using optional name-value pair arguments.

`s = serialport`, without arguments, connects to the serial port using the property settings of your last cleared `serialport` object instance. The retained properties are `Port`, `BaudRate`, `ByteOrder`, `FlowControl`, `StopBits`, `DataBits`, `Parity`, `Timeout`, and `Terminator`. See “Properties” on page 24-324.

Input Arguments

`port` — Serial port name
character vector | string scalar

Serial port name, specified as a character vector or string scalar. Use `serialportlist` to get a list of connected ports.

Example: `"COM2"`

`baudrate` — Baud rate
double

Baud rate for serial communication, specified as a double.

Example: `9600`

Name-Value Pair Arguments

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

You can use Name-Value pairs to set the `DataBits`, `Parity`, `StopBits`, `FlowControl`, `ByteOrder`, and `Timeout` object properties. See "Properties" on page 24-324 for their data types and allowed values.

Example: `"Timeout",30`

Properties

Object Creation Properties

`Port` — Serial port for connection
string

This property is read-only.

Serial port for connection, returned as a string.

Example: `"COM1"`

Data Types: `string`

`BaudRate` — Communication speed
double

Communication speed in bits per second, returned as a positive integer double.

Example: `14400`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Parity` — Parity
`"none"` (default) | `"even"` | `"odd"`

Parity to check whether data has been lost or written, returned as `"none"`, `"even"`, or `"odd"`.

Example: `"odd"`

Data Types: `char` | `string`

`DataBits` — Number of bits to represent one character of data
8 (default) | 7 | 6 | 5

Number of bits to represent one character of data, returned as `8`, `7`, `6`, or `5`.

Example: `8`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`StopBits` — Pattern of bits that indicates the end of a character
1 (default) | `1.5` | 2

Pattern of bits that indicates the end of a character or of the whole transmission, returned as `1`, `1.5`, or `2`.

Example: `1`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`FlowControl` — Mode for managing data transmission rate
`"none"` (default) | `"hardware"` | `"software"`

Mode for managing data transmission rate, returned as `"none"`, `"hardware"`, or `"software"`.

Example: `"software"`

Data Types: `char` | `string`

`ByteOrder` — Sequential order of bytes
`"little-endian"` (default) | `"big-endian"`

Sequential order in which bytes are arranged into larger numerical values, returned as `"little-endian"` or `"big-endian"`. Set this property at object creation using a name-value pair argument. You can also change it after object creation using dot notation.

Example: `"little-endian"`

Data Types: `char` | `string`

`Timeout` — Allowed time to complete operations
10 (default) | numeric

Allowed time in seconds to complete read and write operations, returned as a numeric value. Set this property at object creation using a name-value pair argument. You can also change it after object creation using dot notation.

Example: `60`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Read and Write Properties

`NumBytesAvailable` — Number of bytes available to read
numeric

This property is read-only.

Number of bytes available to read, returned as a numeric value.

Example: `1024`

Data Types: `double`

`NumBytesWritten` — Total number of bytes written to device
0 (default) | numeric

This property is read-only.

Number of bytes written to the serial port, returned as a numeric value.

Example: `512`

Data Types: `double`

`Terminator` — Terminator character for data
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

Terminator character for reading and writing ASCII-terminated data, returned as `"LF"`, `"CR"`, or `"CR/LF"`, or a number from 0 to 255, inclusive. If the read and write terminators are different, `Terminator` is returned as a 1x2 cell array of these values. Set this property with the `configureTerminator` function.

Example: `"CR"`

Data Types: `char` | `string`

Callback Properties

`BytesAvailableFcnMode` — Bytes available callback trigger mode
`"off"` (default) | `"byte"` | `"terminator"`

Bytes available callback trigger mode, returned as `"off"`, `"byte"`, or `"terminator"`. This setting determines if the callback is off, triggered by the number of bytes specified by `BytesAvailableFcnCount`, or triggered by the terminator specified by `Terminator`. Set this property with the `configureCallback` function.

Example: `"off"`

Data Types: `char` | `string`

`BytesAvailableFcnCount` — Number of bytes of data to trigger callback
64 (default) | numeric value

Number of bytes of data to trigger the callback specified by `BytesAvailableFcn`, returned as a double. This value is used only when the `BytesAvailableFcnMode` property is `"byte"`. Set these properties with the `configureCallback` function.

Example: 128

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BytesAvailableFcn` — Callback function triggered by bytes available event
function handle

Callback function triggered by a bytes available event, returned as a function handle. A bytes available event is generated by receiving a certain number of bytes or a terminator. This property is empty until you assign a function handle. Set this property with the `configureCallback` function.

Example: `@myFcn`

Data Types: `function_handle`

`ErrorOccurredFcn` — Callback function triggered by error event
function handle

Callback function triggered by an error event, returned as a function handle. An error event is generated when an asynchronous read or write error occurs. This property is empty until you assign a function handle.

Example: `@myErrFcn`

Data Types: `function_handle`

`UserData` — General purpose property for user data
any type

General purpose property for user data, returned as any MATLAB data type. For example, you can use this property to store data when an event is triggered from a callback function.

Example: `datetime()`

Object Functions

read	Read data from serial port
readline	Read line of ASCII string data from serial port
readbinblock	Read one binblock of data from serial port
write	Write data to serial port
writeline	Write line of ASCII data to serial port
writebinblock	Write one binblock of data to serial port
writeread	Write command to serial port and read response
flush	Clear serial port device buffers
configureTerminator	Set terminator for ASCII string communication with serial port
configureCallback	Set callback function and trigger condition for communication with serial port device
getpinstatus	Get serial pin status
setRTS	Set serial RTS pin
setDTR	Set serial DTR pin

Examples

Read Data from Serial Port

Read 16 values of uint32 data from the `COM3` serial port.

```
s = serialport("COM3",9600,"Timeout",5);
data = read(s,16,"uint32");
```

See Also

Functions
`serialportlist`

Topics
"Read Streaming Data from Arduino Using Serial Port Communication" on page 22-99
"Read Waveform from Tektronix TDS 1002 Scope Using SCPI Commands" on page 22-102

Introduced in R2019b

serialportlist

List of serial ports connected to your system

Syntax

```
serialportlist
serialportlist("all")
serialportlist("available")
```

Description

`serialportlist` or `serialportlist("all")` returns a list of all serial ports on a system. The list includes virtual serial ports provided by USB-to-serial devices and Bluetooth Serial Port Profile devices. The list shows all the serial ports you can access on your computer and can use for serial port communication.

`serialportlist("available")` returns a list of only those serial ports on your system that are available at this time.

Examples

Get List of Serial Ports

Identify serial ports on your system.

```
ports = serialportlist
```

```
ports =

  1×4 string array

    "COM1"    "COM3"    "COM11"    "COM12"
```

List only those ports that are available.

```
freeports = serialportlist("available")
```

```
freeports =

  1×2 string array

    "COM1"    "COM12"
```

See Also

Functions
`serialport`

Introduced in R2019b

set

Configure or display instrument object properties

Syntax

```
set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,PN,PV)
set(obj,S)
```

Arguments

`obj`	An instrument object or an array of instrument objects.
`'PropertyName'`	A property name for `obj`.
`PropertyValue`	A property value supported by *PropertyName*.
`PN`	A cell array of property names.
`PV`	A cell array of property values.
`S`	A structure with property names and property values.
`props`	A structure array whose field names are the property names for `obj`, or cell array of possible values.

Description

`set(obj)` displays all configurable property values for `obj`. If a property has a finite list of possible character vector values, then these values are also displayed.

`props = set(obj)` returns all configurable properties and their possible values for `obj` to `props`. `props` is a structure whose field names are the property names of `obj`, and whose values are cell arrays of possible property values. If the property does not have a finite set of possible values, then the cell array is empty.

`set(obj,'PropertyName')` displays the valid values for *PropertyName* if it possesses a finite list of character vector values.

`props = set(obj,'PropertyName')` returns the valid values for *PropertyName* to `props`. `props` is a cell array of possible character vector values or an empty cell array if *PropertyName* does not have a finite list of possible values.

`set(obj,'PropertyName',PropertyValue,...)` configures multiple property values with a single command.

`set(obj,PN,PV)` configures the properties specified in the cell array of character vectors `PN` to the corresponding values in the cell array `PV`. `PN` must be a vector. `PV` can be m-by-n where m is equal to the number of instrument objects in `obj` and n is equal to the length of `PN`.

`set(obj,S)` configures the named properties to the specified values for `obj`. `S` is a structure whose field names are instrument object properties, and whose field values are the values of the corresponding properties.

Examples

This example illustrates some of the ways you can use `set` to configure or return property values for the GPIB object `g`.

```
g = gpib('ni',0,1);
set(g,'EOSMode','read','OutputBufferSize',50000)
set(g,{'EOSCharCode','RecordName'},{13,'sydney.txt'})
set(g,'EOIMode')
[ {on} | off ]
```

Tips

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to `set`. Additionally, you can specify a property name without regard to case, and you can make use of property name completion. For example, if `g` is a GPIB object, then the following commands are all valid.

```
set(g,'EOSMode')
set(g,'eosmode')
set(g,'EOSM')
```

See Also

`instrhelp` | `propinfo` | `get`

Introduced before R2006a

setDTR

Set serial DTR pin

Syntax

```
setDTR(device,true)
setDTR(device,false)
```

Description

`setDTR(device,true)` sets the specified serial data terminal ready (DTR) pin.

`setDTR(device,false)` resets the specified serial DTR pin.

Examples

Set and Reset DTR Pin

Set and then reset the serial DTR pin.

```
device = serialport("COM3",9600);
  ⋮
setDTR(device,true)
  ⋮
setDTR(device,false)
```

Set and Reset DTR Pin Using VISA

Set and then reset the serial DTR pin using the VISA-Serial interface.

```
device = visadev("COM3");
  ⋮
setDTR(device,true)
  ⋮
setDTR(device,false)
```

Input Arguments

device — Serial port connection
`serialport` object | `visadev` object

Serial port, specified as a `serialport` object or `visadev` object.

Example: `setDTR(device,true)` sets the DTR pin for the serial port connection `device`.

Example: `setDTR(device,true)` sets the DTR pin for the VISA-Serial resource `device`.

See Also

Functions
`serialport` | `visadev` | `setRTS`

Introduced in R2019b

setRTS

Set serial RTS pin

Syntax

```
setRTS(device,true)
setRTS(device,false)
```

Description

`setRTS(device,true)` sets the specified serial request to send (RTS) pin.

`setRTS(device,false)` resets the specified serial RTS pin.

Examples

Set and Reset RTS Pin

Set and then reset the serial RTS pin.

```
device = serialport("COM3",9600);
  ⋮
setRTS(device,true)
  ⋮
setRTS(device,false)
```

Set and Reset RTS Pin Using VISA

Set and then reset the serial RTS pin using the VISA-Serial interface.

```
device = visadev("COM3");
  ⋮
setRTS(device,true)
  ⋮
setRTS(device,false)
```

Input Arguments

device — Serial port connection
`serialport` object | `visadev` object

Serial port, specified as a `serialport` object or `visadev` object.

Example: `setRTS(device,true)` sets the RTS pin for the serial port connection `device`.

Example: `setRTS(device,true)` sets the RTS pin for the VISA-Serial resource `device`.

See Also

Functions
`serialport` | `visadev` | `setDTR`

Introduced in R2019b

size

Size of instrument object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,m3,...,mn] = size(obj)
m = size(obj,dim)
```

Arguments

`obj`	An instrument object or an array of instrument objects.
`dim`	The dimension of `obj`.
`d`	The number of rows and columns in `obj`.
`m`	The number of rows in `obj`, or the length of the dimension specified by `dim`.
`n`	The number of columns in `obj`.
`m1,m2,...,mn`	The length of the first N dimensions of `obj`.

Description

`d = size(obj)` returns the two-element row vector `d` containing the number of rows and columns in `obj`.

`[m,n] = size(obj)` returns the number of rows and columns in separate output variables.

`[m1,m2,m3,...,mn] = size(obj)` returns the length of the first `n` dimensions of `obj`.

`m = size(obj,dim)` returns the length of the dimension specified by the scalar `dim`. For example, `size(obj,1)` returns the number of rows.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

See Also

`instrhelp` | `length`

Introduced before R2006a

spi

Create SPI object

Syntax

```
S = spi(Vendor,BoardIndex,Port);
```

Description

`S = spi(Vendor,BoardIndex,Port);` constructs an `spi` object associated with `Vendor`, `BoardIndex`, and `Port`. `Vendor` must be set to either `'aardvark'`, for use with a Total Phase Aardvark adaptor, or to `'ni845x'`, for use with the NI-845x adaptor board, to use this interface. `BoardIndex` specifies the board index of the hardware and is usually `0`. `Port` specifies the port number within the device and must be set to `0`.

SPI, or Serial Peripheral Interface, is a synchronous serial data link standard that operates in full duplex mode. Instrument Control Toolbox SPI support lets you open connections with individual chips and to read and write over the connections to individual chips using an Aardvark host adaptor.

The primary uses for the `spi` interface involve the `write`, `read`, and `writeAndRead` functions for synchronously reading and writing binary data. To identify SPI devices in the Instrument Control Toolbox, use the `instrhwinfo` function on the SPI interface, called `spi`.

Once the SPI object is created, there are properties that can be used to change communication settings. These includes properties for clock speed, clock phase, and clock polarity. For a list of all the properties and information about setting them, see the link for “Using Properties on the SPI Object” at the end of the Examples section.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Examples

Communicate With SPI Device

This example shows how to create a SPI object and communicate with a SPI device, using an Aardvark adaptor board.

Ensure that the Aardvark adaptor is installed so that you can use the `spi` interface, and then look at the adaptor properties.

```
instrhwinfo('spi')
instrhwinfo('spi', 'aardvark')
```

```
ans =
```

```
          VendorName: 'aardvark'
   VendorDescription: 'Total Phase I2C/SPI Driver'
   VendorLibraryName: 'aardvark.dll'
    InstalledBoardIds:  {[0]}
  BoardSerialNumbers:  {'2237722838'}
  ObjectConstructors:  {'spi('aardvark', 0, 0)'}
```

Construct a `spi` object called `S` using `Vendor 'aardvark'`, with `BoardIndex` of `0`, and `Port` of `0`.

```
S = spi('aardvark', 0, 0);
```

You can optionally change property settings such as `BitRate`, `ClockPhase`, or `ClockPolarity`. For example, set the `ClockPhase` from the default of `FirstEdge`.

```
S.ClockPhase = 'SecondEdge'
```

For a list of all the properties and information about setting them, see the link for “Using Properties on the SPI Object” at the end of the Examples section.

Connect to the chip.

```
connect(S);
```

Read and write to the chip.

```
% Create a variable containing the data to write
dataToWrite = [3 0 0 0];

% Write the binary data to the chip
write(S, dataToWrite);

% Create a variable to contain 5 bytes of returned data
numData = 5

% Read the binary data from the chip
read(S, numData)
```

Disconnect the SPI device and clean up by clearing the object.

```
disconnect(S);
clear('S');
```

Input Arguments

Vendor — Adaptor board vendor
`'aardvark'` | `'ni845x'`

Adaptor board vendor, specified as the character vector `'aardvark'` or `'ni845x'`. You need to use a Total Phase Aardvark adaptor or an NI-845x adaptor board to use the SPI interface. You must specify this as the first argument when you create the `spi` object.

Example: `S = spi('aardvark', 0, 0);`

Data Types: `char` | `string`

BoardIndex — Board index of your hardware
`0`

Board index of your hardware, specified as a numeric value. This is usually `0`. You must specify this as the second argument when you create the `spi` object.

Example: `S = spi('aardvark', 0, 0);`

Data Types: `double`

`Port` — Port number of your hardware

`0`

Port number of your hardware, specified as the number `0`. The Aardvark adaptor uses `0` as the port number. You must specify this as the third argument when you create the `spi` object.

Example: `S = spi('aardvark', 0, 0);`

Data Types: `double`

See Also

Topics

"Using Properties on the SPI Object" on page 10-13

Introduced in R2013b

spoll

(To be removed) Perform serial poll on GPIB objects

Note This gpib object function will be removed in a future release. Use visadev object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
out = spoll(obj)
out = spoll(obj,val)
[out] = spoll(obj)
[out,statusByte] = spoll(obj)
[out] = spoll(obj,val)
[out,statusByte] = spoll(obj,val)
```

Arguments

obj	A GPIB object or an array of GPIB objects.
val	A numeric array containing the indices of the objects in obj, that must be ready for servicing before control is returned to the MATLAB Command Window.
out	The GPIB objects ready for servicing.
statusByte	The service request (SRQ) line status byte.

Description

out = spoll(obj) performs a serial poll on the instruments associated with obj. out contains the GPIB objects that are ready for servicing. If no objects are ready for servicing, then out is empty.

out = spoll(obj,val) performs a serial poll and waits until the instruments specified by val are ready for servicing. An error is returned if a value specified in val does not match an index value in obj.

Using this syntax, spoll blocks access to the MATLAB Command Window until the objects specified by val are ready for servicing, or a timeout occurs for each object specified by val. The timeout period is specified by the Timeout property.

[out] = spoll(obj) returns the object or an array of objects.

[out,statusByte] = spoll(obj) returns the status byte along with the object or an array of objects.

[out] = spoll(obj,val) returns the object and the value specified in the index value of the object.

[out,statusByte] = spoll(obj,val) returns the status byte along with the object and the value specified in the index value of the object.

Examples

If `obj` is a four-element array and `val` is set to `[1 3]`, then `spoll` will block access to the MATLAB Command Window until the instruments connected to the first and third GPIB objects have both asserted their SRQ line, or a timeout occurs.

Example of second output argument:

```
g1 = gpib('ni', 0, 1);
g2 = gpib('ni', 0, 2);
fopen([g1 g2]);
out1 = spoll(g1);
out2 = spoll([g1 g2], 1);
out3 = spoll([g1 g2], [1 2])
[out4 statusBytes] = spoll([g1 g2])
[out5 statusBytes] = spoll([g1 g2], 2)
fclose([g1 g2]);
```

Tips

Serial polling is a method of obtaining specific information from GPIB objects when they request service. When you perform a serial poll, `out` contains the GPIB object that has asserted its SRQ line.

If `obj` is an array of GPIB objects

- Each element of `obj` must have the same `BoardIndex` property value.
- Each element of `obj` is polled to determine if the instrument is ready for servicing.

If you specify a second output argument when you call an `spoll`, full serial poll bytes are returned in addition to the SRQ line status in the second argument.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Compatibility Considerations

`gpib` object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

`visastatus` | `gpib` | `length` | `spoll (visa)` | BoardIndex | Timeout

Introduced before R2006a

start

Enables RF signal generator signal output and modulation output

Syntax

`start(rf, CenterFrequency, OutputPower, LoopCount)`

Description

`start(rf, CenterFrequency, OutputPower, LoopCount)` enables signal output and modulation output for the RF signal generator `rf`. All three arguments are required.

Examples

Start RF Signal Generator Signal and Modulation Output

Use the `start` function on an RF signal generator object to start signal output and modulation output. It takes a double value for each of the three required arguments: `CenterFrequency` specified in Hz, `OutputPower` specified in dBm, and `LoopCount`, which represents the number of times the waveform should be repeated.

Create an `rfsiggen` object to communicate with an RF signal generator, using the VISA resource string and driver associated with your own instrument.

```
rf = rfsiggen('TCPIP0::172.28.22.99::inst0::INSTR','AgRfSigGen')
```

When you designate the `Resource` and `Driver` properties during object creation, it automatically connects to the instrument.

Assign the `CenterFrequency`, `OutputPower`, and `LoopCount` variables to use in the signal generation.

```
CenterFrequency = 4000000
OutputPower = 0
LoopCount = inf
```

Start the signal generation.

```
start(rf, CenterFrequency, OutputPower, LoopCount)
```

Input Arguments

`CenterFrequency` — Center frequency for the waveform
double

Center frequency for the waveform, specified as a double. This value, in Hz, should be the first argument after the object name.

Example: `start(rf, CenterFrequency, OutputPower, LoopCount)`

Data Types: `double`

`OutputPower` — Output power for the RF signal generation
double

Output power for the RF signal generation, specified as a double. This value, in dBm, should be the second argument after the object name.

Example: `start(rf, CenterFrequency, OutputPower, LoopCount)`

Data Types: `double`

`LoopCount` — Number of times to repeat waveform
double

Number of times to repeat waveform, specified as a double. This value should be the third argument after the object name.

Example: `start(rf, CenterFrequency, OutputPower, LoopCount)`

Data Types: `double`

See Also

`rfsiggen` | `download` | `resources` | `drivers`

Topics

"Download and Generate Signals with RF Signal Generator" on page 14-45
"Quick-Control RF Signal Generator Functions" on page 14-41
"Quick-Control RF Signal Generator Properties" on page 14-43

Introduced in R2017b

spoll (visa)

(To be removed) Perform serial poll on VISA objects

Note This `visa` object function will be removed in a future release. Use `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
out = spoll(obj)
out = spoll(obj,val)
[out] = spoll(obj)
[out,statusByte] = spoll(obj)
[out] = spoll(obj,val)
[out,statusByte] = spoll(obj,val)
```

Arguments

`obj`	A VISA object or an array of VISA objects.
`val`	A numeric array containing the indices of the objects in `obj`, that must be ready for servicing before control is returned to the MATLAB Command Window.
`out`	The VISA objects ready for servicing.
`statusByte`	The service request (SRQ) line status byte.

Description

`out = spoll(obj)` performs a serial poll on the instruments associated with `obj`. `out` contains the VISA objects that are ready for servicing. If no objects are ready for servicing, then `out` is empty.

`out = spoll(obj,val)` performs a serial poll and waits until the instruments specified by `val` are ready for servicing. An error is returned if a value specified in `val` does not match an index value in `obj`.

Using this syntax, `spoll` blocks access to the MATLAB Command Window until the objects specified by `val` are ready for servicing, or a timeout occurs for each object specified by `val`. The timeout period is specified by the `Timeout` property.

`[out] = spoll(obj)` returns the object or an array of objects.

`[out,statusByte] = spoll(obj)` returns the status byte along with the object or an array of objects.

`[out] = spoll(obj,val)` returns the object and the value specified in the index value of the object.

`[out,statusByte] = spoll(obj,val)` returns the status byte along with the object and the value specified in the index value of the object.

Examples

If `obj` is a four-element array and `val` is set to `[1 3]`, then `spoll` will block access to the MATLAB Command Window until the instruments connected to the first and third VISA objects have both requested servicing, or a timeout occurs.

Example of second output argument:

```
v1 = visa('keysight', 'TCPIP0::yourdomainname.com::inst0::INSTR');
v2 = visa('keysight', 'TCPIP0::yourdomainname.com::inst01::INSTR');
fopen([v1 v2]);
out1 = spoll(v1);
out2 = spoll([v1 v2], 1);
out3 = spoll([v1 v2], [1 2])
[out4 statusBytes] = spoll([v1 v2])
[out5 statusBytes] = spoll([v1 v2], 2)
fclose([v1 v2]);
```

Tips

Serial polling is a method of obtaining specific information from VISA objects when they request service. When you perform a serial poll, `out` contains the VISA object that have requested servicing.

If `obj` is an array of VISA objects

- Each element of `obj` must have the same `BoardIndex` property value.
- Each element of `obj` is polled to determine if the instrument is ready for servicing.

If you specify a second output argument when you call an `spoll`, full serial poll bytes are returned in addition to the SRQ line status in the second argument.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Compatibility Considerations

`visa` object interface will be removed

Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

See Also

`visastatus` | `visa` | `spoll` | BoardIndex | Timeout

Introduced in R2010a

stopasync

(To be removed) Stop asynchronous read and write operations

Note This `serial`, `Bluetooth`, `tcpip`, `udp`, `visa`, and `gpib` object function will be removed in a future release. Use `serialport`, `bluetooth`, `tcpclient`, `tcpserver`, `udpport`, and `visadev` object functions instead. For more information, see “Compatibility Considerations”.

Syntax

```
stopasync(obj)
```

Arguments

`obj`	An interface object or an array of interface objects.

Description

`stopasync(obj)` stops any asynchronous read or write operation that is in progress for `obj`.

Tips

You can write data asynchronously using the `fprintf` or `fwrite` functions. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous` (serial port, TCPIP, UDP, and VISA-serial objects). In-progress asynchronous operations are indicated by the `TransferStatus` property.

If `obj` is an array of interface objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops,

- Its `TransferStatus` property is configured to `idle`.
- Its `ReadAsyncMode` property is configured to `manual` (serial port, TCPIP, UDP, and VISA-serial objects).
- The data in its output buffer is flushed.

Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the `readasync` function, or configure the `ReadAsyncMode` property to `continuous`, then the new data is appended to the existing data in the input buffer.

Compatibility Considerations

`serial` object interface will be removed

Not recommended starting in R2019b

Use of this function with a `serial` object will be removed. To access a serial port device, use a `serialport` object with its functions and properties instead.

See “Transition Your Code to serialport Interface” on page 6-26 for more information about using the recommended functionality.

`Bluetooth` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `Bluetooth` object will be removed. To access a Bluetooth device, use a `bluetooth` object with its functions and properties instead.

See “Transition Your Code to bluetooth Interface” for more information about using the recommended functionality.

`tcpip` object interface will be removed
Not recommended starting in R2020b

Use of this function with a `tcpip` object will be removed. To create a TCP/IP client or server, use a `tcpclient` or `tcpserver` object with its functions and properties instead.

See “Transition Your Code to tcpclient Interface” on page 7-36 or “Transition Your Code to tcpserver Interface” on page 7-42 for more information about using the recommended functionality.

udp object interface will be removed
Not recommended starting in R2020b

Use of this function with a `udp` object will be removed. To access a UDP socket, use a `udpport` object with its functions and properties instead.

See “Transition Your Code to udpport Interface” on page 8-18 for more information about using the recommended functionality.

`visa` object interface will be removed
Not recommended starting in R2021a

Use of this function with a `visa` object will be removed. To access a VISA resource, use a `visadev` object with its functions and properties instead.

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

`gpib` object interface will be removed
Not recommended starting in R2021b

Use of this function with a `gpib` object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a `visadev` object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

Functions
`fprintf` | `fwrite` | `readasync`

Properties
ReadAsyncMode | TransferStatus

Introduced before R2006a

tcpclient

Create TCP/IP client connection with TCP/IP server

Description

A tcpclient object represents a connection to a remote host and remote port from MATLAB to read and write data. The remote host can be a server or hardware that supports TCP/IP communication, and must already exist. The tcpclient object is always the client and cannot be used as a server. For information on creating a TCP/IP server, see "Communicate Using TCP/IP Server Sockets" on page 7-34.

Creation

Syntax

```
t = tcpclient(address,port)
t = tcpclient(address,port,Name,Value)
```

Description

t = tcpclient(address,port) creates a TCP/IP client that connects to a server associated with the remote host address and remote port port. The value of address can be either a remote host name or a remote host IP address. The value of port must be a number between 1 and 65535. The input address sets the Address property and the input port sets the Port property.

If you specified an invalid address or port, the TCP/IP server is not running, or the connection to the server cannot be established, then the object is not created and MATLAB throws an error.

t = tcpclient(address,port,Name,Value) creates a connection and sets additional "Properties" using one or more name-value pair arguments. Set the Timeout and ConnectTimeout properties using name-value pair arguments. Enclose each property name in quotes, followed by the property value.

Example: t = tcpclient("144.212.130.17",80,"Timeout",20,"ConnectTimeout",30) creates a TCP/IP client connection to the TCP/IP server on port 80 at IP address 44.212.130.17. It sets the timeout period to 20 seconds and the connection timeout to 30 seconds.

Properties

Object Creation Properties

Address — Remote host name or IP address
character vector | string scalar

Remote host name or IP address, specified as a character vector or string scalar. This property can be set only at object creation.

Example: t = tcpclient("www.mathworks.com",80) creates a TCP/IP client connection to port 80 at www.mathworks.com.

Example: `t = tcpclient("144.212.130.17",80)` creates a TCP/IP client connection to the TCP/IP server on port 80 at IP address 144.212.130.17.

Data Types: `char` | `string`

`Port` — Remote host port
numeric

Remote host port, specified as a number between 1 and 65535, inclusive. This property can be set only at object creation.

Example: `t = tcpclient("www.mathworks.com",80)` creates a TCP/IP client connection to port 4012 at www.mathworks.com.

Data Types: `double`

`Timeout` — Allowed time to complete operations
10 (default) | numeric

Allowed time in seconds to complete read and write operations, specified as a numeric value. Set this property at object creation using a name-value pair argument. You can also change it after object creation using dot notation.

Example: `t = tcpclient("144.212.130.17",80,"Timeout",20)` sets the read/write timeout period to 20 seconds.

Data Types: `double`

`ConnectTimeout` — Allowed time to connect to remote host
`Inf` (default) | numeric

Allowed time in seconds to connect to the remote host, specified as a numeric value. This property specifies the maximum time to wait for a connection request to the specified remote host to succeed or fail. This property can be set only at object creation.

Example: `t = tcpclient("144.212.130.17",80,"ConnectTimeout", 30)` sets the connection timeout period to 30 seconds.

Data Types: `double`

Read and Write Properties

`NumBytesAvailable` — Number of bytes available to read
numeric

This property is read-only.

Number of bytes available to read, returned as a numeric value.

Example: `t.NumBytesAvailable` returns the number of bytes available to read.

Data Types: `double`

`NumBytesWritten` — Total number of bytes written to remote host
0 (default) | numeric

This property is read-only.

Total number of bytes written to the remote host, returned as a numeric value.

Example: `t.NumBytesWritten` returns the number of bytes written.

Data Types: `double`

`ByteOrder` — Sequential order of bytes
`"little-endian"` (default) | `"big-endian"`

Sequential order in which bytes are arranged into larger numerical values, specified as `"little-endian"` or `"big-endian"`.

Set the value of this property when reading and writing multi-byte data types, such as `uint16`, `int16`, `uint32`, `int32`, `single`, or `double`. The value of this property must match the configuration of the remote host connected to `tcpclient`. The remote host or other applications might have a default byte order of big-endian, while the default value of this property is `little-endian`.

Example: `t.ByteOrder = "big-endian"` sets the byte order to big-endian.

Data Types: `char` | `string`

`Terminator` — Terminator character for data
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

Terminator character for reading and writing ASCII-terminated data, returned as `"LF"`, `"CR"`, `"CR/LF"`, or a number from 0 to 255, inclusive. If the read and write terminators are different, `Terminator` is returned as a 1x2 cell array of these values. Set this property with the `configureTerminator` function.

Example: `configureTerminator(t,"CR")` sets both the read and write terminators to `"CR"`.

Example: `configureTerminator(t,"CR",10)` sets the read terminator to `"CR"` and the write terminator to `10`.

Data Types: `double` | `char` | `string`

Callback Properties

`BytesAvailableFcnMode` — Bytes available callback trigger mode
`"off"` (default) | `"byte"` | `"terminator"`

Bytes available callback trigger mode, returned as `"off"`, `"byte"`, or `"terminator"`. This setting determines if the callback is off, triggered by the number of bytes specified by `BytesAvailableFcnCount`, or triggered by the terminator specified by `Terminator`. Set this property with the `configureCallback` function.

Example: `configureCallback(t,"byte",50,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 50 bytes of new data are available to be read.

Example: `configureCallback(t,"terminator",@callbackFcn)` sets the `callbackFcn` callback to trigger when a terminator is available to be read.

Example: `configureCallback(dev,"off")` turns off callbacks.

Data Types: `char` | `string`

`BytesAvailableFcnCount` — Number of bytes of data to trigger callback
64 (default) | numeric

Number of bytes of data to trigger the callback specified by `BytesAvailableFcn`, returned as a double. This value is used only when the `BytesAvailableFcnMode` property is `"byte"`. Set these properties with the `configureCallback` function.

Example: `configureCallback(t,"byte",50,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 50 bytes of new data are available to be read.

Data Types: `double`

`BytesAvailableFcn` — Callback function triggered by bytes available event
function handle

Callback function triggered by a bytes available event, returned as a function handle. A bytes available event is generated by receiving a certain number of bytes or a terminator. This property is empty until you assign a function handle. Set this property with the `configureCallback` function.

Example: `configureCallback(t,"byte",50,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 50 bytes of new data are available to be read.

Data Types: `function_handle`

`ErrorOccurredFcn` — Callback function triggered by error event
function handle

Callback function triggered by an error event, returned as a function handle. An error event is generated when an asynchronous read or write error occurs. This property is empty until you assign a function handle.

Example: `t.ErrorOccurredFcn = @myErrorFcn`

Data Types: `function_handle`

`UserData` — General purpose property for user data
any type

General purpose property for user data, returned as any MATLAB data type. For example, you can use this property to store data when an event is triggered from a callback function.

Example: `t.UserData`

Object Functions

read	Read data from remote host over TCP/IP
readline	Read line of ASCII string data from remote host over TCP/IP
readbinblock	Read one binblock of data from remote host over TCP/IP
write	Write data to remote host over TCP/IP
writeline	Write line of ASCII data to remote host over TCP/IP
writebinblock	Write one binblock of data to remote host over TCP/IP
writeread	Write command to remote host over TCP/IP and read response
configureTerminator	Set terminator for ASCII string communication with remote host over TCP/IP
configureCallback	Set callback function and trigger condition for communication with remote host over TCP/IP
flush	Clear buffers for communication with remote host over TCP/IP

Examples

Connect to TCP/IP Remote Host Using Host Name

Create the TCP/IP object `t` using the host address shown and port `80`.

```
t = tcpclient("www.mathworks.com",80)
```

```
t = 
  tcpclient with properties:

              Address: 'www.mathworks.com'
                 Port: 80
    NumBytesAvailable: 0

  Show all properties, functions
```

When you connect using a host name, such as a specified web address or `'localhost'`, the IP address defaults to IPv6 format. If the server you are connecting to is expecting IPv4 format, connection fails. For IPv4, you can create a connection by specifying an explicit IP address rather than a host name.

Connect to TCP/IP Remote Host Using IP Address

Create a TCP/IP client connection called `t` using the IP address shown and port `80`.

```
t = tcpclient("144.212.130.17",80)
```

```
t = 
  tcpclient with properties:

              Address: '144.212.130.17'
                 Port: 80
    NumBytesAvailable: 0

  Show all properties, functions
```

Connect to TCP/IP Remote Host and Set Timeout Period

Create a TCP/IP client connection called `t` and set the timeout period to 20 seconds.

```
t = tcpclient("144.212.130.17",80,"Timeout",20)
```

```
t = 
  tcpclient with properties:

              Address: '144.212.130.17'
                 Port: 80
    NumBytesAvailable: 0

  Show all properties, functions

ans = 20
```

See the value of `Timeout`.

```
t.Timeout
```

The output reflects the property change.

Connect to TCP/IP Remote Host and Set Connection Timeout Period

Create a TCP/IP client connection called `t` and set the `ConnectTimeout` property to 30 seconds.

```
t = tcpclient("144.212.130.17",80,"ConnectTimeout",30)
```

```
t = 
  tcpclient with properties:

              Address: '144.212.130.17'
                 Port: 80
    NumBytesAvailable: 0

  Show all properties, functions
```

See the value of `ConnectTimeout`.

```
t.ConnectTimeout
```

```
ans = 30
```

The output reflects the property change.

Write and Read uint8 Data from Remote Host

Create a TCP/IP client connection called `t`, connecting to a TCP/IP echo server with port 4000. To do so, you must have an `echotcpip` server running on port 4000.

```
echotcpip("on",4000)
t = tcpclient("localhost",4000)
```

```
t = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

The `write` function synchronously writes data to the remote host connected to `t`. First specify the data and then write the data. The function suspends MATLAB execution until the specified number of values is written to the remote host.

Assign 10 bytes of `uint8` data to the variable `data`.

```
data = uint8(1:10)
```

```
data = 1×10 uint8 row vector
```

```
    1    2    3    4    5    6    7    8    9   10
```

View the data.

```
whos data
```

```
  Name      Size            Bytes  Class    Attributes

  data      1x10               10  uint8
```

Write data to the echo server.

```
write(t,data)
```

Confirm the success of the writing operation by viewing the `NumBytesAvailable` property.

```
t.NumBytesAvailable
```

```
ans = 10
```

Since the client is connected to an echo server, the data you write to the server is returned to the client. Read all the bytes of data available.

```
read(t)
```

```
ans = 1×10 uint8 row vector

    1    2    3    4    5    6    7    8    9   10
```

Using the `read` function with no arguments reads all available bytes of data from `t` connected to the remote host and returns the data. The number of values read is determined by the `NumBytesAvailable` property, which is the number of bytes available in the input buffer.

Close the connection between the TCP/IP client and the remote host by clearing the object. Turn off the `echotcpip` server.

```
clear t
echotcpip("off")
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Platform-specific code generation is only supported on desktop platforms (Windows, macOS, Linux) and on Raspberry Pi™.
- The following `tcpclient` properties do not support code generation:
  - `NumBytesAvailable`
  - `NumBytesWritten`
  - `ByteOrder`

  - `Terminator`
  - `BytesAvailableFcnMode`
  - `BytesAvailableFcnCount`
  - `BytesAvailableFcn`
  - `ErrorOccurredFcn`
  - `UserData`

  Only `Address`, `Port`, `Timeout`, and `ConnectTimeout` are supported.
- The following `tcpclient` object functions do not support code generation:
  - `readline`
  - `readbinblock`
  - `writeline`
  - `writebinblock`
  - `writeread`
  - `configureTerminator`
  - `configureCallback`
  - `flush`

Only `read` and `write` are supported.

See Also
`tcpserver` | `echotcpip`

Topics
"Create TCP/IP Client and Configure Settings" on page 7-3
"Write and Read Data over TCP/IP Interface" on page 7-7

Introduced in R2014b

tcpip

(To be removed) Create TCPIP object

Note `tcpip` and its object properties will be removed. Use `tcpclient` or `tcpserver` and its properties instead. For more information, see “Compatibility Considerations”.

Syntax

```
t = tcpip(RemoteHost)
t = tcpip(RemoteHost,RemotePort)
t = tcpip( ___ ,Name,Value)
```

Description

`t = tcpip(RemoteHost)` creates a TCPIP object, `t`, associated with remote host `RemoteHost` and the default remote port value of 80.

When the TCPIP object is created, its `Status` property value is `closed`. When the object is connected to the host with the `fopen` function, the `Status` property is configured to `open`.

The default local host in multihome hosts is the system default. The `LocalPort` property defaults to a value of `[]`, which allows any free local port to be used. `LocalPort` is assigned when `fopen` is issued.

`t = tcpip(RemoteHost,RemotePort)` creates a TCPIP object with the specified remote port value `RemotePort`.

`t = tcpip( ___ ,Name,Value)` creates a TCPIP object with the specified optional name-value pairs. If an invalid property name or property value is specified, the object is not created.

Examples

Write and Read with a TCP/IP Echo Server

Use a TCPIP object to write to an echo server and read back the message.

Start a TCP/IP echo server and create a TCPIP object.

```
echotcpip('on',4012)
t = tcpip('localhost',4012);
```

Connect the TCPIP object to the host.

```
fopen(t)
```

Write to the host and read from the host.

```
fwrite(t,65:74)
A = fread(t,10)
```

```
A =

    65
    66
    67
    68
    69
    70
    71
    72
    73
    74
```

Disconnect the TCPIP object from the host and stop the echo server.

```
fclose(t)
echotcpip('off')
```

Input Arguments

`RemoteHost` — Remote host ID
char vector | string

Remote host ID, specified as a character vector or string, identifying IP address or host name.

Example: `'127.0.0.1'`

Data Types: `char` | `string`

`RemotePort` — Port on remote host
80 (default) | numeric value

Port on remote host, specified as a numeric integer value from 1 to 65535.

Example: `8001`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

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

Example: `'NetworkRole','server'`

`NetworkRole` — Network role of TCPIP interface object
`'client'` (default) | `'server'`

Network role for TCPIP object, specified as `'client'` or `'server'`. For example, `t = tcpip('localhost',30000,'NetworkRole','server')` creates a TCPIP object, `t`, that is an interface for a server socket.

Example: `'server'`

Data Types: `char` | `string`

`Name` — Name of interface object
char | string

Name of interface object, specified as a character vector or string.

Example: 'TCPdev1'

Data Types: char | string

Timeout — Time limit for communication
numeric

Time limit in seconds for communication, specified as a numeric value. The default is 10 seconds.

Example: 60

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Output Arguments

t — TCPIP interface
interface object

TCPIP interface, returned as an interface object.

Compatibility Considerations

tcpip function will be removed
Not recommended starting in R2020b

tcpip and its object properties will be removed. Use tcpclient or tcpserver and its properties instead.

This example shows how to create a TCP/IP client using the recommended functionality.

Functionality	Use This Instead
`t = tcpip("127.0.0.1",3030,"NetworkRole","client");` `fopen(t)`	`t = tcpclient("127.0.0.1",3030);`

This example shows how to create a TCP/IP server using the recommended functionality.

Functionality	Use This Instead

`t = tcpip("192.168.2.15",3030,"NetworkRole","server");`
`fopen(t)`
 	
This binds to host "0.0.0.0" (internally) and port 3030 and only accepts client connections coming from "192.168.2.15". MATLAB is blocked until a client connection is established.
	`t = tcpserver("0.0.0.0",3030);`
	
	This binds to "0.0.0.0" and port 3030. "0.0.0.0" means that it accepts any incoming client connection requests on port 3030. MATLAB is not blocked.

See “Transition Your Code to tcpclient Interface” on page 7-36 or “Transition Your Code to tcpserver Interface” on page 7-42 for more information about using the recommended functionality.

See Also

Functions
echotcpip | fclose | fopen | fread | fwrite

Properties
LocalHost | LocalPort | LocalPortMode | Name | RemoteHost | RemotePort | Status | Type

Introduced before R2006a

tcpserver

Create TCP/IP server

Description

A `tcpserver` object represents a TCP/IP server that receives a TCP/IP client connection request from the specified IP address and port number and accepts the request. Once the server establishes a connection, you can receive data from and send data to the client using read and write functions. Each `tcpserver` object supports only one client connection at a time.

Creation

Syntax

```
t = tcpserver(address,port)
t = tcpserver(port)
t = tcpserver(___,Name,Value)
```

Description

`t = tcpserver(address,port)` creates a TCP/IP server that listens for a TCP/IP client connection request at the IP address specified by `address` and the port number specified by `port`.

The input argument `address` sets the `ServerAddress` property and the input argument `port` sets the `ServerPort` property.

`t = tcpserver(port)` creates a TCP/IP server that listens for a client connection request at port number `port` and IP address `"::"`. This IP address indicates that the server accepts a client connection from any valid IP address on the machine.

`t = tcpserver(___,Name,Value)` creates a TCP/IP server and sets additional "Properties" on page 24-360 using one or more name-value pair arguments. Set the `Timeout`, `ByteOrder`, and `ConnectionChangedFcn` properties using name-value pair arguments. After any of the input argument combinations in the previous syntaxes, enclose each property name in quotes, followed by the property value.

For example, `t = tcpserver(4000,"Timeout",20,"ByteOrder","big-endian")` creates a TCP/IP server that listens for connections on port `4000` at the IP address `"::"`. It sets the timeout period to 20 seconds and the byte order to big-endian.

Properties

Object Creation Properties

`ServerAddress` — IP address where server listens
`"::"` (default) | character vector | string scalar

IP address where the server listens for TCP/IP client connections, specified as a character vector or string scalar. You can set this property to any valid IPV4 address, IPV6 address, or host name of the machine. This property can be set only at object creation.

Example: `t = tcpserver("144.212.100.10",4000)` listens for connections at port 4000 and IP address 144.212.100.10.

Note If you specify a host name at object creation, `tcpserver` resolves it to an IPV4 or IPV6 address and sets `ServerAddress` to the resolved IP address.

Data Types: `char` | `string`

`ServerPort` — Port number where server listens
numeric

Port number where the server listens for TCP/IP client connections, specified as a number between 1 and 65535, inclusive. This property can be set only at object creation.

Example: `t = tcpserver("144.212.100.10",4000)` listens for connections at port 4000 and IP address 144.212.100.10.

Data Types: `double`

`Timeout` — Allowed time to complete read and write operations
10 (default) | numeric

Allowed time in seconds to complete read and write operations, specified as a numeric value. Set this property at object creation using a name-value pair argument. You can also change it after object creation using dot notation.

Example: `t = tcpserver("144.212.100.10",4000,"Timeout",20)` sets the read/write timeout period to 20 seconds.

Data Types: `double`

`ByteOrder` — Sequential order of bytes
`"little-endian"` (default) | `"big-endian"`

Sequential order in which bytes are arranged into larger numerical values, specified as `"little-endian"` or `"big-endian"`. This only applies for the following numeric data types: `uint16`, `int16`, `uint32`, `int32`, `uint64`, `int64`, `single`, and `double`. Set this property at object creation using a name-value pair argument. You can also change it after object creation using dot notation.

Example: `t = tcpserver("144.212.100.10",4000,"ByteOrder","big-endian")` sets the byte order to big-endian.

Data Types: `char` | `string`

`ConnectionChangedFcn` — Callback function triggered by connection or disconnection event
function handle

Callback function triggered by connection or disconnection event, specified as a function handle. A connection or disconnection event occurs when a TCP/IP client connects to or disconnects from the server. Set this property at object creation using a name-value pair argument. You can also change it after object creation using dot notation. This property is empty until you assign a function handle.

Example: `t = tcpserver("144.212.100.10",4000,"ConnectionChangedFcn",@myConnectionFcn)` sets the connection callback function to `myConnectionFcn`. When a client connects or disconnects, `myConnectionFcn` triggers.

Data Types: `function_handle`

Connection Properties

`Connected` — Server connection status
`false` or `0` (default) | `true` or `1`

This property is read-only.

Server connection status, returned as a numeric or logical `1` (`true`) or `0` (`false`). If the value of this property is `true`, a TCP/IP client is connected to the server.

You can connect to only one client at a time. If a client disconnects from the server, you can connect to another client immediately.

Data Types: `logical`

`ClientAddress` — IP address of connected client
`""` (default) | string scalar

This property is read-only.

IP address of the connected client, returned as a string. The value of this property matches the IP address of the client. The value of this property is empty until a TCP/IP client establishes a connection to the server. If a client disconnects from the server, the value of this property becomes empty.

Example: `t.ClientAddress` returns the IP address of the connected client.

Data Types: `string`

`ClientPort` — Port number of connected client
`[]` (default) | numeric

This property is read-only.

Port number of the connected client, returned as a double. The value of this property is empty until a TCP/IP client establishes a connection to the server.

Example: `t.ClientPort` returns the port number of the connected client.

Data Types: `double`

Read and Write Properties

`Terminator` — Terminator character for data
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

Terminator character for reading and writing ASCII-terminated data, returned as `"LF"`, `"CR"`, `"CR/LF"`, or a numeric integer from 0 to 255, inclusive. If the read and write terminators are different, `Terminator` is returned as a 1x2 cell array of these values. Set this property with the `configureTerminator` function.

Example: `configureTerminator(t,"CR")` sets both the read and write terminators to `"CR"`.

Example: `configureTerminator(t,"CR",10)` sets the read terminator to `"CR"` and the write terminator to `10`.

Data Types: `double` | `char` | `string`

`NumBytesAvailable` — Number of bytes available to read
0 (default) | numeric

This property is read-only.

Number of bytes available to read, returned as a numeric value.

Example: `t.NumBytesAvailable` returns the number of bytes available to read.

Data Types: `double`

`NumBytesWritten` — Total number of bytes written
0 (default) | numeric

This property is read-only.

Total number of bytes written, returned as a numeric value. The value of this property does not reset to 0 when a client disconnects or reconnects to the server.

Example: `t.NumBytesWritten` returns the number of bytes written.

Data Types: `double`

Callback Properties

`BytesAvailableFcnMode` — Bytes available callback trigger mode
`"off"` (default) | `"byte"` | `"terminator"`

Bytes available callback trigger mode, returned as `"off"`, `"byte"`, or `"terminator"`. This setting determines if the callback is off, triggered by the number of bytes specified by `BytesAvailableFcnCount`, or triggered by the terminator specified by `Terminator`. Set this property with the `configureCallback` function.

Example: `configureCallback(t,"byte",50,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 50 bytes of new data are available to be read.

Example: `configureCallback(t,"terminator",@callbackFcn)` sets the `callbackFcn` callback to trigger when a terminator is available to be read.

Example: `configureCallback(t,"off")` turns off callbacks.

Data Types: `char` | `string`

`BytesAvailableFcnCount` — Number of bytes of data to trigger callback
64 (default) | numeric

Number of bytes of data to trigger the callback specified by `BytesAvailableFcn`, returned as a double. This value is used only when the `BytesAvailableFcnMode` property is `"byte"`. Set these properties with the `configureCallback` function.

Example: `configureCallback(t,"byte",50,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 50 bytes of new data are available to be read.

Data Types: `double`

`BytesAvailableFcn` — Callback function triggered by bytes available event
function handle

Callback function triggered by a bytes available event, returned as a function handle. A bytes available event is generated by receiving a certain number of bytes or a terminator. This property is empty until you assign a function handle. Set this property with the `configureCallback` function.

Example: `configureCallback(t,"byte",50,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 50 bytes of new data are available to be read.

Data Types: `function_handle`

`ErrorOccurredFcn` — Callback function triggered by error event
function handle

Callback function triggered by an error event, returned as a function handle. An error event is generated when the network connection for the server is interrupted or lost. This property is empty until you assign a function handle.

Example: `t.ErrorOccurredFcn = @myErrorFcn`

Data Types: `function_handle`

`UserData` — General purpose property for user data
any type

General purpose property for user data, returned as any MATLAB data type. For example, you can use this property to store data from a callback function.

Example: `t.UserData`

Object Functions

read	Read data sent to TCP/IP server
readline	Read line of ASCII string data sent to TCP/IP server
readbinblock	Read one binblock of data sent to TCP/IP server
write	Write data from TCP/IP server
writeline	Write line of ASCII data from TCP/IP server
writebinblock	Write one binblock of data from TCP/IP server
configureTerminator	Set terminator for ASCII string communication
configureCallback	Set callback function and trigger condition for communication
flush	Clear buffers for communication using TCP/IP server

Examples

Create TCP/IP Server that Listens at IP Address and Port Number

Create a TCP/IP server called `t` that listens for connections at your machine's IP address and port 4000. Your IP address is different from the one in this example. It must be any valid IPV4 address, IPV6 address, or host name of the adapter on the machine.

```
t = tcpserver("172.28.200.145",4000)
```

```
t =
  TCPServer with properties:
```

```
        ServerAddress: "172.28.200.145"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

The values of the `Connected`, `ClientAddress`, and `ClientPort` properties indicate that a TCP/IP client is not connected to the server.

Create TCP/IP Server that Listens at Port Number

Create a TCP/IP server called `t` that listens for connections at all IP addresses and port 4000.

```
t = tcpserver(4000)
```

```
t = 
  TCPServer with properties:

        ServerAddress: "::"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

The values of the `Connected`, `ClientAddress`, and `ClientPort` properties indicate that a TCP/IP client is not connected to the server.

Create TCP/IP Server and Set Timeout Period

Create a TCP/IP server called `t` and set the read and write timeout period to 20 seconds.

```
t = tcpserver(4000,"Timeout",20)
```

```
t = 
  TCPServer with properties:

        ServerAddress: "::"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Display the value of `Timeout`.

```
t.Timeout
```

```
ans = 20
```

The output shows the specified timeout value, indicating that `t` waits for up to 20 seconds to complete a read or write operation.

Create TCP/IP Server and Set Connection Event Callback

Create a callback function called `connectionFcn` and save it as a `.m` file in the current working directory. When this callback function is invoked, it displays a message in the MATLAB Command Window indicating connection or disconnection. You can modify this code to perform read or write operations on your TCP/IP server instead of displaying a message.

```
function connectionFcn(src,~)
if src.Connected
   disp("This message is sent by the server after accepting the client connection request.")
else
   disp("Client has disconnected.")
end
end
```

Create the TCP/IP server called `server` and set the `ConnectionChangedFcn` property to a handle to the `connectionFcn` callback function.

```
server = tcpserver("localhost",4000,"ConnectionChangedFcn",@connectionFcn)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client called `client` with the same IP address and port number as your server.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

```
This message is sent by the server after accepting the client connection
request.
```

After you create the client, it connects to the server. This triggers a connection event for the server, which invokes the `connectionFcn` callback function. The callback function returns the message you see in the Command Window.

Disconnect the client from the server by clearing it.

```
clear client
```

```
Client has disconnected.
```

Clearing the client triggers a disconnection event for the server and returns the message from the `connectionFcn` callback function.

Write String Data from TCP/IP Server

Create a TCP/IP server that listens for a client connection request at the specified port and IP address. Then, write data from the server to the connected client.

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

See the values of the `Connected`, `ClientAddress`, and `ClientPort` properties for `server`.

```
server
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 1
        ClientAddress: "127.0.0.1"
           ClientPort: 65136
    NumBytesAvailable: 0

  Show all properties, functions
```

The output shows that `server` successfully accepts a request from `client` and that `client` establishes a connection to `server`.

Send data to the client by writing it using the `server` object. Since the client is connected to the server, this data is available in the client. Read this data from the `client` object.

```
write(server,"hello world","string")
read(client,11,"string")
```

```
ans = 
"hello world"
```

Read String Data Sent to TCP/IP Server

Create a TCP/IP server that listens for a client connection request at the specified port and IP address. Then read data sent to the server from the connected client.

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
```

```
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Display the values of the `Connected`, `ClientAddress`, and `ClientPort` properties for `server`.

```
server
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 1
        ClientAddress: "127.0.0.1"
           ClientPort: 65440
    NumBytesAvailable: 0

  Show all properties, functions
```

The output shows that `server` successfully accepts a request from `client` and that `client` establishes a connection to `server`.

Write data to the TCP/IP client. Since the client is connected to the server, this data is available in the server. Read the first five values of string data using the `server` object.

```
write(client,"helloworld","string")
read(server,5,"string")
```

```
ans = 
"hello"
```

If you read five more values, you receive the remaining string data.

```
read(server,5,"string")
```

```
ans = 
"world"
```

See Also

`tcpclient` | `echotcpip`

Topics

“Communicate Using TCP/IP Server Sockets” on page 7-34
“Communicate Between a TCP/IP Client and Server in MATLAB” on page 22-127

Introduced in R2021a

trigger

(To be removed) Send trigger message to instrument

Note This visa and gpib object function will be removed in a future release. Use visadev object functions instead. For more information, see "Compatibility Considerations".

Syntax

```
trigger(obj)
```

Arguments

obj	A GPIB, VISA-GPIB, or VISA-VXI object.

Description

trigger(obj) sends a trigger message to the instrument connected to obj.

Tips

Before you can use trigger, obj must be connected to the instrument with the fopen function. A connected interface object has a Status property value of open. An error is returned if you attempt to use trigger while obj is not connected to the instrument.

For GPIB and VISA-GPIB objects, the Group Execute Trigger (GET) message is sent to the instrument.

For VISA-VXI objects, if the TriggerType property is configured to software, the Word Serial Trigger command is sent to the instrument. If TriggerType is configured to hardware, a hardware trigger is sent on the line specified by the TriggerLine property.

Compatibility Considerations

visa object interface will be removed

Not recommended starting in R2021a

Use of this function with a visa object will be removed. To access a VISA resource, use a visadev object with its functions and properties instead.

See "Transition Your Code to visadev Interface" on page 5-32 for more information about using the recommended functionality.

gpib object interface will be removed

Not recommended starting in R2021b

Use of this function with a gpib object will be removed. To access a GPIB instrument, use the VISA-GPIB interface with a visadev object, its functions, and its properties instead.

See “Transition Your Code to VISA-GPIB Interface” on page 4-16 for more information about using the recommended functionality.

See Also

`visatrigger` | `fopen` | Status | TriggerLine | TriggerType

Introduced before R2006a

udp

(To be removed) Create UDP object

Note `udp` will be removed in a future release. Use `udpport` instead. For more information, see "Compatibility Considerations".

Syntax

```
u = udp
u = udp(RemoteHost)
u = udp(RemoteHost,RemotePort)
u = udp( ___ ,Name,Value)
```

Description

`u = udp` creates a UDP object, `u`, not associated with a remote host. If you use this syntax, you must assign a remote host after object creation if you want to send data. If you want to only receive data, you do not need to set a remote host.

The UDP object must be bound to the local socket with the `fopen` function. The default local host in multihome hosts is the system default. The `LocalPort` property defaults to a value of `[]`, allowing any free local port to be used. `LocalPort` is updated with a value when `fopen` is issued. When the UDP object is created, its `Status` property value is `'closed'`. Once the object is bound to the local socket with `fopen`, `Status` is configured to `'open'`.

The maximum packet size for reading is 8192 bytes. The input buffer can hold as many packets as defined by the `InputBufferSize` property value. You can write any data size to the output buffer. The data is sent in packets of at most 4096 bytes.

`u = udp(RemoteHost)` creates a UDP object associated with the remote host `RemoteHost`.

`u = udp(RemoteHost,RemotePort)` creates a UDP object with the specified remote port value, `RemotePort`. If not specified, the default remote port is 9090.

`u = udp( ___ ,Name,Value)` creates a UDP object and specifies additional options with one or more name-value pair arguments. If you specify an invalid property name or property value, the object is not created.

Examples

Write and Read with a UDP Echo Server

Use a UDP object to write to an echo server and read back the message.

Start the echo server and create a UDP object.

```
echoudp('on',4012)
u = udp('127.0.0.1',4012);
```

Connect the UDP object to the host.

```
fopen(u)
```

Write to the host, and then read from the host.

```
fwrite(u,65:74)
A = fread(u,10)
```

```
A =

    65
    66
    67
    68
    69
    70
    71
    72
    73
    74
```

Stop the echo server and disconnect the UDP object from the host.

```
echoudp('off')
fclose(u)
```

Input Arguments

`RemoteHost` — Remote host ID
char vector | string

Remote host ID, specified as a character vector or string, identifying IP address or host name.

Example: `'127.0.0.1'`

Data Types: `char` | `string`

`RemotePort` — Port on remote host
9090 (default) | numeric value

Port on remote host, specified as a numeric integer value from 1 to 65535.

Example: `8001`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

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

Example: `'LocalPort',4080`

Commonly used properties for this object are:

`LocalPort` — Port on local host
numeric value

Port on local host, specified as a numeric integer value from 1 to 65535.

Example: `4080`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`LocalPortMode` — Port mode on local host
`'auto'` (default) | `'manual'`

Port mode on local host, specified as `'manual'` or `'auto'`.

Example: `'manual'`

Data Types: `char` | `string`

`Timeout` — Time limit for communication
numeric

Time limit in seconds for communication, specified as a numeric value. The default is 10 seconds.

Example: `60`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Output Arguments

u — UDP interface
interface object

UDP interface, returned as an interface object.

Compatibility Considerations

udp function will be removed
Not recommended starting in R2020b

`udp` and its object properties will be removed in a future release. Use `udpport` and its properties instead.

This example shows how to connect to a UDP socket using the recommended functionality.

Functionality	Use This Instead
`u = udp;` `fopen(u)`	`u = udpport("byte");`
	`u = udpport("datagram");`

See “Transition Your Code to udpport Interface” on page 8-18 for more information about using the recommended functionality.

See Also

Functions
`udpport` | `echoudp` | `fclose` | `fopen` | `fread` | `fwrite`

Properties
LocalHost | LocalPort | LocalPortMode | Name | RemoteHost | RemotePort | Status | Type

Introduced before R2006a

udpport

Connect to UDP socket

Description

The `udpport` object allows you to perform byte-type and datagram-type UDP communication using a UDP socket on the local host.

Creation

Syntax

```
u = udpport
u = udpport("byte")
u = udpport(IPv)
u = udpport("byte",IPv)
u = udpport("datagram")
u = udpport("datagram",IPv)
u = udpport( ___ ,Name,Value)
```

Description

`u = udpport` or `u = udpport("byte")` constructs a byte-type `udpport` object `u`, with an IP address version set to IPV4, bound to a UDP socket.

`u = udpport(IPv)` or `u = udpport("byte",IPv)` constructs a byte-type `udpport` object with an IP address version specified by `IPv`, which can be either `"IPV4"` or `"IPV6"`.

`u = udpport("datagram")` constructs a datagram-type `udpport` object `u`, with an IP address version set to IPV4.

`u = udpport("datagram",IPv)` constructs a datagram-type udpport object with an IP address version specified by `IPv`, which can be either `"IPV4"` or `"IPV6"`.

`u = udpport( ___ ,Name,Value)` constructs a `udpport` object and sets specified object properties using name-value pair arguments. If you specify an invalid property name or value, the function does not create the object. `udpport` properties that can be set using name-value pair arguments are `LocalHost`, `LocalPort`, `Timeout`, `ByteOrder`, `OutputDatagramSize`, and `EnablePortSharing`.

Input Arguments

IPv — IP address version
`"IPV4"` (default) | `"IPV6"`

IP address version, specified as `"IPV4"` or `"IPV6"`.

Note The IP address version must be consistent across your setup. You cannot communicate between an IPV4 socket and IPV6 address, or vice versa.

Name-Value Pair Arguments

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

Properties that support Name-Value arguments are `LocalHost`, `LocalPort`, `Timeout`, `ByteOrder`, `OutputDatagramSize`, and `EnablePortSharing`. Of these, `LocalHost`, `LocalPort`, and `EnablePortSharing` become read-only after the `udpport` object is created.

Example: `"Timeout",60`

Properties

Properties Common to Byte and Datagram `udpport` Objects

`IPAddressVersion` — Version type for IP address
`"IPV4"` (default) | `"IPV6"`

This property is read-only.

Version type for IP address, specified as `"IPV4"` or `"IPV6"`.

Example: `"IPV6"`

Data Types: `char` | `string`

`LocalHost` — Local host name or IP address
string | character vector

This property is read-only.

Local host name or dotted-decimal IP address, specified as a character vector or string. If you do not specify a value for `LocalHost` when you create the `udpport` object, the default value is `"0.0.0.0"` for IPV4, or `"::"` for IPV6.

Example: `"144.133.0.0"`

Data Types: `char` | `string`

`LocalPort` — Port of local host for binding
numeric

This property is read-only.

Port of local host for binding for UDP, specified as a numeric value from 0 to 65535. If you do not specify a value for `LocalPort` when you create the `udpport` object, a value is automatically assigned.

Example: `50791`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`ByteOrder` — Sequential arrangement of bytes
`"little-endian"` (default) | `"big-endian"`

Sequential order in which bytes are arranged into larger numerical values, specified as `"little-endian"` or `"big-endian"`.

Example: "big-endian"

Data Types: char | string

Timeout — Time allowed for read and write operations
10 (default) | double

Time allowed for read and write operations in seconds, specified as a double.

Example: 20

Data Types: double

OutputDatagramSize — Maximum number of bytes written in datagram packet
512 (default) | double

Maximum number of bytes of data to be written in a datagram packet, specified as a double value from 1 to 65507.

Example: 512

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

EnablePortSharing — Allow other UDP sockets to bind to same local port
false (0) (default) | true (1)

This property is read-only.

Setting to allow other UDP sockets to bind to the same local port as this socket, specified as logical true (1) or false (0).

Example: true

Data Types: logical

EnableBroadcast — Allow broadcasting
false (0) (default) | true (1)

Setting to allow broadcasting, specified as logical true (1) or false (0).

Example: true

Data Types: logical

EnableMulticast — Allow multicast
false (0) (default) | true (1)

This property is read-only.

Setting to allow multicast, specified as logical true (1) or false (0).

Example: true

Data Types: logical

MulticastGroup — IP address group to receive multicast data
string

This property is read-only.

IP address group to subscribe to for receiving multicast data. Set this property with the `configureMulticast` function.

Example: `"226.0.0.1"`

Data Types: `char` | `string`

`EnableMulticastLoopback` — Allow looping back if sender is in multicast group
`true` (1) (default) | `false` (0)

This property is read-only.

Indicate looping back of data in `udpport` multicast if the sender is subscribed to the same multicast group, specified as a logical. Set this property with the `configureMulticast` function.

Example: `false`

Data Types: `logical`

`ErrorOccurredFcn` — Function to call when error event occurs
function handle

Function to call when an error event occurs, specified as a function handle.

Example: `@myErrorFun`

Data Types: `function_handle`

`UserData` — Application-specific data
any type

Application-specific data for this `udpport` instance. This is a general purpose property for user data, specified as any MATLAB data type. For example, you can use this property to store data when an event is triggered from a callback function.

Example: `datetime()`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `datetime` | `duration` | `calendarDuration` | `fi`

Byte `udpport` Object Properties

`NumBytesAvailable` — Number of bytes available to be read
double

This property is read-only.

Number of bytes available to be read, returned as a double.

Example: `512`

Data Types: `double`

`NumBytesWritten` — Number of bytes written to `udpport` socket
double

This property is read-only.

Number of bytes written to `udpport` socket, returned as a double.

Example: 48

Data Types: `double`

`Terminator` — Terminator for ASCII string communication
`"LF"` (default) | `"CR"` | `"CR/LF"` | numeric

This property is read-only.

Terminator for ASCII-terminated string communication, returned as a string or numeric value. If the read and write operations have different terminators, the values are retuned as a cell array. To set this property value, use the `configureTerminator` function.

Example: `"CR/LF"`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `string` | `cell`

`BytesAvailableFcn` — Function called when bytes-available event occurs
function handle

This property is read-only.

Function called when a bytes-available event occurs, specified as a function handle. To set this property value, use the `configureCallback` function.

Example: `@myAvailFun`

Data Types: `function_handle`

`BytesAvailableFcnCount` — Number of available bytes to trigger event
numeric

This property is read-only.

Number of bytes required in the input buffer to trigger a bytes-available event, returned as a numeric value. This property applies only when `BytesAvailableFcnMode` is `"byte"`. To set this property value, use the `configureCallback` function.

Example: 64

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`BytesAvailableFcnMode` — Condition for triggering `BytesAvailableFcn` callback
`"off"` (default) | `"byte"` | `"terminator"`

This property is read-only.

Condition for triggering `BytesAvailableFcn` callback, returned as `"off"`, `"byte"`, or `"terminator"`. To set this property value, use the `configureCallback` function.

Example: `"byte"`

Data Types: `string`

Datagram `udpport` Object Properties

`NumDatagramsAvailable` — Number of datagrams available to be read
double

This property is read-only.

Number of datagrams available to be read, returned as a double.

Example: 64

Data Types: double

NumDatagramsWritten — Number of datagrams written to udpport socket
double

This property is read-only.

Number of datagrams written to udpport socket, returned as a double.

Example: 8

Data Types: double

DatagramsAvailableFcn — Function called when datagrams-available event occurs
function handle

This property is read-only.

Function called when a datagrams-available event occurs, returned as a function handle. To set this property value, use the configureCallback function.

Example: @myAvailFcn

Data Types: function_handle

DatagramsAvailableFcnCount — Number of available datagrams to trigger event
numeric

This property is read-only.

Number of available datagrams to trigger a datagrams-available event, returned as a numeric value. This property applies only when BytesAvailableFcnMode is "datagram". To set this property value, use the configureCallback function.

Example: 16

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

DatagramsAvailableFcnMode — Condition for triggering DatagramsAvailableFcn callback
"off" (default) | "datagram"

This property is read-only.

Condition for triggering a DatagramsAvailableFcn callback, returned as "off" or "datagram". To set this property value, use the configureCallback function.

Example: "datagram"

Data Types: string

Object Functions

Functions for byte-type and datagram-type udpport interfaces:

read	Read data from UDP socket
write	Write data to UDP socket
configureCallback	Set callback function and trigger condition for communication with UDP socket
configureMulticast	Set multicast properties for communication with UDP socket
flush	Clear UDP socket buffers

Functions for a byte-type `udpport` interface only:

readline	Read line of ASCII string data from UDP socket
writeline	Write line of ASCII data to UDP socket
configureTerminator	Set terminator for ASCII string communication with UDP socket

Examples

Send and Receive UDP Bytes

This example shows common tasks of byte-type UDP communication.

Construct a byte-type `udpport` object.

```
u = udpport("IPV4")
```

```
u = 

  UDPPort with properties:

     IPAddressVersion: "IPV4"
            LocalHost: "0.0.0.0"
            LocalPort: 60825
    NumBytesAvailable: 0
```

Write a vector of `uint8` data via the `udpport` socket to a specified address and port..

```
write(u,1:5,"uint8","125.0.1.4",2020);
```

Read 10 values of `uint16` data from the `udpport` socket.

```
data = read(u,10,"uint16");
```

Define the terminator, and send a string via the `udpport` socket to a specified address and port.

```
configureTerminator(u,"CR/LF");
writeline(u,"hello","125.0.1.4",2020);
```

Read an ASCII-terminated string from the `udpport` socket.

```
data = readline(u);
```

Subscribe to a multicast address group.

```
configureMulticast(u,"226.0.0.1");
```

Configure a callback to trigger when 50 bytes are available.

```
configureCallback(u,"byte",50,@myCallbackFcn);
```

Flush the output buffer, then disconnect and clear the `udpport` connection.

```
flush(u,"output");
clear u
```

Send and Receive UDP Datagrams

This example shows common tasks of datagram-type UDP communication.

Construct a datagram-type `udpport` object.

```
u = udpport("datagram","IPV4")
```

```
u = 

  UDPPort with properties:

         IPAddressVersion: "IPV4"
                LocalHost: "0.0.0.0"
                LocalPort: 53465
    NumDatagramsAvailable: 0
```

Write a vector of `uint8` data via the udpport socket to a specified address and port.

```
write(u,1:5,"uint8","125.0.1.4",2020);
```

Read one datagram packet as `uint16` data.

```
data = read(u,1,"uint16");
```

Subscribe to a multicast address group.

```
configureMulticast(u,"226.0.0.1");
```

Configure a callback to trigger when 5 datagrams are available.

```
configureCallback(u,"datagram",5,@myCallbackFcn);
```

Flush the output buffer, then disconnect and clear the `udpport` connection.

```
flush(u,"output");
clear u
```

Control Datagram Size

This example shows how datagram size affects data segmentation.

Turn `echoudp` on at port 3030, then create a datagram-type `udpport` object with an `OutputDatagramSize` of 5.

```
echoudp("on",3030);
u = udpport("datagram","OutputDatagramSize",5);
```

Send 20 bytes of `uint8` data to the `echoudp` port.

```
write(u,1:20,"uint8","127.0.0.1",3030);
```

Because `OutputDatagramSize` is set to 5, the 20 bytes are sent as 4 datagram packets, each containing 5 bytes of data.

Verify that 4 datagrams were received from the echo server.

```
u.NumDatagramsAvailable
```

```
ans =

     4
```

Read the 4 datagrams received from the echo server.

```
data = read(u,u.NumDatagramsAvailable,"uint8")
```

```
data = 

  1×4 Datagram array with properties:

    Data
    SenderAddress
    SenderPort
```

The first datagram contains the values 1-5 (5 bytes), the second 6-10, the third 11-15, and the fourth 16-20.

View the third datagram.

```
data(3)
```

```
ans = 

  Datagram with properties:

             Data: [11 12 13 14 15]
    SenderAddress: "127.0.0.1"
       SenderPort: 3030
```

Allow UDP Port Sharing

Allow multiple `udpport` objects to share the same local port.

Create a `udpport` object bound to `LocalPort` 3030.

```
u1 = udpport("LocalPort",3030,"EnablePortSharing",true);
```

Create a separate `udpport` object using the same port.

```
u2 = udpport("LocalPort",3030,"EnablePortSharing",true);
```

`EnablePortSharing` must be `true` for both `udpport` objects.

See Also

Functions
`echoudp`

Introduced in R2020b

update

Update entry of IVI configuration store object

Syntax

```
update(obj, 'type', 'name', 'P1', V1, ...)
update(obj, struct)
```

Arguments

`obj`	IVI configuration store object.
`'type'`	Type of entry; *type* can be `HardwareAsset`, `DriverSession`, or `LogicalName`.
`'name'`	Name of the `DriverSession`, `HardwareAsset`, or `LogicalName` to be updated.
`'P1'`	First parameter for updated entry; other parameter-value pairs may follow.
`V1`	Value for first parameter.
`struct`	Structure defining entry fields to be updated.

Description

`update(obj, 'type', 'name', 'P1', V1, ...)` updates an entry of type, *type*, with name, `name`, in IVI configuration store object, `obj`, using the specified parameter-value pairs. *type* can be `HardwareAsset`, `DriverSession`, or `LogicalName`.

If an entry of type, *type* with name, `name` does not exist, an error will occur.

Valid parameters for a `DriverSession` are listed below. The default value for on/off parameters is `off`.

Parameter	Value	Description
`Name`	character vector	A unique name for the driver session.
`SoftwareModule`	character vector	The name of a software module entry in the configuration store.
`HardwareAsset`	character vector	The name of a hardware asset entry in the configuration store.
`Description`	Any character vector	Description of driver session
VirtualNames	structure	A struct array containing virtual name mappings
`Cache`	`on/off`	Enable caching if the driver supports it.
`DriverSetup`	Any character vector	This value is software module dependent

Parameter	Value	Description
`InterchangeCheck`	`on/off`	Enable driver interchangeability checking, if supported
`QueryInstrStatus`	`on/off`	Enable instrument status querying by the driver
`RangeCheck`	`on/off`	Enable extended range checking by the driver, if supported
`RecordCoercions`	`on/off`	Enable recording of coercions by the driver, if supported
`Simulate`	`on/off`	Enable simulation by the driver

Valid fields for `HardwareAsset` are

Parameter	Value	Description
`Name`	character vector	A unique name for the hardware asset
`Description`	Any character vector	Description of hardware asset
`IOResourceDescriptor`	character vector	The I/O address of the hardware asset

Valid fields for `LogicalName` are

Parameter	Value	Description
Name	character vector	A unique name for the logical name
`Description`	Any character vector	Description of hardware asset
Session	character vector	The name of a driver session entry in the configuration store

`update(obj, struct)` updates the entry using the fields in `struct`. If an entry with the type and name field in `struct` does not exist, an error will occur. Note that the `name` field cannot be updated using this syntax.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see “Using Tab Completion for Functions” on page 2-4.

Examples

Update the `Description` parameter of the driver session named `ScopeSession` in the IVI configuration store object named `c`.

```
c = iviconfigurationstore;
update(c, 'DriverSession', 'ScopeSession', 'Description', ...
'A session.');
```

See Also

`iviconfigurationstore` | `add` | `commit` | `remove`

Introduced before R2006a

visa

(To be removed) Create VISA object

Note `visa` will be removed in a future release. Use `visadev` instead. For more information, see "Compatibility Considerations".

Syntax

```
obj = visa('vendor','rsrcname')
```

Arguments

'*vendor*'	A supported VISA vendor.
'rsrcname'	The resource name of the VISA instrument.
'*PropertyName*'	A VISA property name.
PropertyValue	A property value supported by *PropertyName*.
obj	The VISA object.

Description

`obj = visa('`*`vendor`*`','rsrcname')` creates the VISA object `obj` with a resource name given by `rsrcname` for the vendor specified by *`vendor`*.

You must first configure your VISA resources in the vendor's tool first, and then you create these VISA objects. Use `instrhwinfo` to find the commands to configure the objects:

```
vinfo = instrhwinfo('visa','keysight');
vinfo.ObjectConstructorName
```

If an invalid vendor or resource name is specified, an error is returned and the VISA object is not created. For a list of supported values for *`vendor`* see Supported Vendor and Resource Names. on page 5-3

Examples

Create a VISA-serial object connected to serial port COM1 using National Instruments VISA interface.

```
vs = visa('ni','ASRL1::INSTR');
```

Create a VISA-GPIB object connected to board 0 with primary address 1 and secondary address 30 using Keysight VISA interface.

```
vg = visa('keysight','GPIB0::1::30::INSTR');
```

Create a VISA-VXI object connected to a VXI instrument located at logical address 8 in the first VXI chassis.

```
vv = visa('keysight','VXI0::8::INSTR');
```

Create a VISA-GPIB-VXI object connected to a GPIB-VXI instrument located at logical address 72 in the second VXI chassis.

```
vgv = visa('keysight','GPIB-VXI1::72::INSTR');
```

Create a VISA-RSIB object connected to an instrument configured with IP address 192.168.1.33.

```
vr = visa('ni', 'RSIB::192.168.1.33::INSTR')
```

Create a VISA-TCPIP object connected to an instrument configured with IP address 216.148.60.170.

```
vt = visa('tek', 'TCPIP::216.148.60.170::INSTR')
```

Create a VISA-USB object connected to a USB instrument with manufacturer ID 0x1234, model code 125, and serial number A22-5.

```
vu = visa('keysight', 'USB::0x1234::125::A22-5::INSTR')
```

Tips

At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with VISA objects.

```
instrhelp visa
```

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, the following commands are all valid.

```
v = visa('ni','GPIB0::1::INSTR','SecondaryAddress', 96);
v = visa('ni','GPIB0::1::INSTR','secondaryaddress', 96);
v = visa('ni','GPIB0::1::INSTR','SECOND', 96);
```

Before you can communicate with the instrument, it must be connected to `obj` with the `fopen` function. A connected VISA object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while `obj` is not connected to the instrument. You cannot connect multiple VISA objects to the same instrument.

Creating a VISA-GPIB Object

When you create a VISA-GPIB object, these properties are automatically configured:

- `Type` is given by `visa-gpib`.
- `Name` is given by concatenating `VISA-GPIB` with the board index, the primary address, and the secondary address.
- `BoardIndex`, `PrimaryAddress`, `SecondaryAddress`, and `RsrcName` are given by the values specified during object creation.

Creating a VISA-GPIB-VXI Object

When you create a VISA-GPIB-VXI object, these properties are automatically configured:

- `Type` is given by `visa-gpib-vxi`.
- `Name` is given by concatenating `VISA-GPIB-VXI` with the chassis index and the logical address specified in the `visa` function.
- `ChassisIndex`, `LogicalAddress`, and `RsrcName` are given by the values specified during object creation.
- `BoardIndex`, `PrimaryAddress`, and `SecondaryAddress` are given by the `visa` driver after the object is connected to the instrument with `fopen`.

Creating a VISA-RSIB Object

When you create a VISA-RSIB object, these properties are automatically configured:

- `Type` is given by `visa-rsib`.
- `Name` is given by concatenating `VISA-RSIB` with the remote host specified in the `visa` function.
- `RemoteHost` and `RsrcName` are given by the values specified during object creation.

Creating a VISA-Serial Object

When you create a VISA-serial object, these properties are automatically configured:

- `Type` is given by `visa-serial`.
- `Name` is given by concatenating `VISA-Serial` with the port specified in the `visa` function.
- `Port` and `RsrcName` are given by the values specified during object creation.

Creating a VISA-TCPIP Object

When you create a VISA-TCPIP object, these properties are automatically configured:

- `Type` is given by `visa-tcpip`.
- `Name` is given by concatenating `VISA-TCPIP` with the board index, remote host, and LAN device name specified in the `visa` function.
- `BoardIndex`, `RemoteHost`, `LANNAme`, and `RsrcName` are given by the values specified during object creation.

Creating a VISA-USB Object

When you create a VISA-USB object, these properties are automatically configured:

- `Type` is given by `visa-usb`.
- `Name` is given by concatenating `VISA-USB` with the board index, manufacturer ID, model code, serial number, and interface number specified in the `visa` function.
- `BoardIndex`, `ManufacturerID`, `ModelCode`, `SerialNumber`, `InterfaceIndex`, and `RsrcName` are given by the values specified during object creation.

Creating a VISA-VXI Object

When you create a VISA-VXI object, these properties are automatically configured:

- `Type` is given by `visa-vxi`.
- `Name` is given by concatenating `VISA-VXI` with the chassis index and the logical address specified in the `visa` function.

- ChassisIndex, LogicalAddress, and RsrcName are given by the values specified during object creation.

Compatibility Considerations

visa function will be removed
Not recommended starting in R2021a

visa and its object properties will be removed in a future release. Use visadev and its properties instead.

This example shows how to connect to a VISA device using the recommended functionality.

Functionality	Use This Instead
v = visa('ni','GPIB0::1::0::INSTR'); fopen(v)	v = visadev('GPIB0::1::0::INSTR');

See “Transition Your Code to visadev Interface” on page 5-32 for more information about using the recommended functionality.

See Also

visadev | fclose | fopen | instrhelp | instrhwinfo | BoardIndex | ChassisIndex | InterfaceIndex | LANName | LogicalAddress | ManufacturerID | ModelCode | Name | Port | PrimaryAddress | RsrcName | SecondaryAddress | SerialNumber | Status | Type

Introduced before R2006a

visadev

Create connection to device using VISA

Description

A `visadev` object represents a connection to a device or instrument using the VISA interface. The following interface types are supported: TCP/IP (using VXI11 and HiSLIP), TCP/IP Socket, USB, GPIB, Serial, VXI, and PXI. Identify devices available to connect to using `visadevlist`. Then, connect to the device or instrument using `visadev`.

Creation

Syntax

```
v = visadev(resourceName)
v = visadev(resourceAlias)
```

Description

`v = visadev(resourceName)` creates a connection to a device using its VISA resource name. Establish a connection using an installed VISA driver. If you have multiple VISA drivers installed, MATLAB uses the preferred VISA set in your VISA vendor's configuration utility software.

`v = visadev(resourceAlias)` creates a connection to a device using its VISA alias, if it has one. If the configuration utility does not yet recognize the device, you cannot connect using the alias and must use the resource name.

Input Arguments

`resourceName` — VISA resource name
character vector | string scalar

VISA resource name, specified as a character vector or string scalar. You can identify the name of the resource you want to connect to using the information returned by `visadevlist`. This input sets the ResourceName property.

Example: `gpibdev = visadev("GPIB0::5::INSTR")` connects to the GPIB device specified by the VISA resource name `GPIB0::5::INSTR`.

Data Types: `char` | `string`

`resourceAlias` — VISA alias associated with resource
character vector | string scalar

VISA alias associated with a resource, specified as a character vector or string scalar. Identify the alias of the resource you want to connect to using the information returned by `visadevlist`. You can use an alias only if an alias is assigned using the VISA vendor’s configuration utility software. This input sets the Alias property.

Example: `serialdev = visadev("COM4")` connects to the serial device specified by the VISA resource alias `COM4`.

Data Types: `char` | `string`

Properties

See visadev Properties for a full list of properties.

Object Functions

read	Read data from VISA resource
readline	Read line of ASCII string data from VISA resource
readbinblock	Read one binblock of data from VISA resource
write	Write data to VISA resource
writeline	Write line of ASCII data to VISA resource
writebinblock	Write one binblock of data to VISA resource
writeread	Write command to VISA resource and read response
configureTerminator	Set terminator for ASCII string communication with VISA resource
configureCallback	Set callback function and trigger condition for communication with VISA resource
flush	Clear buffers for communication with VISA resource
visastatus	Check status of VISA resource
visatrigger	Send trigger message to GPIB or VXI instruments
setDTR	Set serial DTR pin
setRTS	Set serial RTS pin
getpinstatus	Get serial pin status

Examples

Connect to VISA Resource Using Name or Alias

Search for and establish a connection to your VISA resource.

Search for available VISA resources.

```
resourceList = visadevlist
```

```
resourceList =

  6×6 table

                    ResourceName                         Alias                  Vendor
         __________________________________________    _________________    ____________________

    1    "USB0::0x0699::0x036A::CU010105::0::INSTR"    "NI_SCOPE_4CH"       "TEKTRONIX"
    2    "TCPIP0::169.254.2.20::inst0::INSTR"          "Keysight_33210A"    "Agilent Technologies
    3    "ASRL1::INSTR"                                "COM1"               ""
    4    "ASRL3::INSTR"                                "COM3"               ""
    5    "GPIB0::5::INSTR"                             "FGEN_2CH"           "Agilent Technologies
    6    "GPIB0::11::INSTR"                            "OSCOPE_2CH"         "TEKTRONIX"
```

Create a connection to the first resource over the VISA-USB interface using the resource name.

```
usbdev = visadev("USB0::0x0699::0x036A::CU010105::0::INSTR")
```

```
usbdev = 

  USB with properties:

         ResourceName: "USB0::0x0699::0x036A::CU010105::0::INSTR"
                Alias: "NI_SCOPE_4CH"
               Vendor: "TEKTRONIX"
                Model: "TDS 2024B"
    NumBytesAvailable: 0

  Show all properties, functions
```

Alternatively, you can connect to a device using its alias.

```
serialdev = visadev("COM1")
```

```
serialdev = 

  Serial with properties:

         ResourceName: "ASRL1::INSTR"
                Alias: "COM1"
                 Port: "ASRL1"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

See Also
`visadevlist`

Introduced in R2021a

visadevlist

List available VISA resources

Syntax

```
resourceList = visadevlist
resourceList = visadevlist("Timeout",time)
```

Description

`resourceList = visadevlist` returns a table containing information about available VISA resources using an installed VISA driver. If you have multiple drivers installed, MATLAB uses the preferred VISA set in your VISA vendor's configuration utility software. The following interface types are supported: TCP/IP (using VXI11 and HiSLIP), TCP/IP Socket, USB, GPIB, Serial, VXI, and PXI. Use `visadev` to connect to a device.

`resourceList = visadevlist("Timeout",time)` returns all VISA resources found within the specified period, in seconds. The default timeout period is 10 seconds. The value of `time` must be 2 or greater.

Examples

List Available VISA Resources

Use `visadevlist` to list all available VISA resources on your machine.

```
resourceList = visadevlist
```

```
resourceList = 

  6×6 table

                     ResourceName                         Alias                 Vendor
         _______________________________________    __________________    _____________________

    1    "USB0::0x0699::0x036A::CU010105::0::INSTR"    "NI_SCOPE_4CH"       "TEKTRONIX"
    2    "TCPIP0::169.254.2.20::inst0::INSTR"          "Keysight_33210A"    "Agilent Technologies
    3    "ASRL1::INSTR"                                "COM1"               ""
    4    "ASRL3::INSTR"                                "COM3"               ""
    5    "GPIB0::5::INSTR"                             "FGEN_2CH"           "Agilent Technologies
    6    "GPIB0::11::INSTR"                            "OSCOPE_2CH"         "TEKTRONIX"
```

ResourceName and Alias identify each resource, and you can use either one as an input when you create a visadev object. The Vendor, Model, and SerialNumber columns provide additional information about the instrument or device. The Type column contains the type of VISA interface.

Output Arguments

resourceList — List of VISA resources
table

List of VISA resources, returned as a table. The table has the following columns.

ResourceName — VISA resource name
string scalar

VISA resource name, returned as a string scalar. The resource name identifies information about the resource, such as interface type, address, board, or port number. Use this name as an input for visadev to create a connection to your VISA resource.

Alias — VISA alias associated with resource
string scalar

VISA alias associated with a resource, specified as a character vector or string scalar. If an alias has been assigned to a VISA resource using the vendor's configuration utility software, it appears here. You can use this name as an input for visadev to create a connection to your VISA resource.

Vendor — Instrument manufacturer
string scalar

Instrument manufacturer, returned as a character vector or string scalar. This value is empty if the VISA interface type does not provide information about the manufacturer.

Model — Instrument model
string scalar

Instrument model, returned as a character vector or string scalar. This value is empty if the VISA interface type does not provide information about the model.

SerialNumber — Unique serial number associated with instrument
string scalar

Unique serial number associated with an instrument, returned as a character vector or string scalar. This value is empty if the VISA interface type does not provide information about the serial number.

Type — Type of VISA resource
gpib | pxi | serial | socket | tcpip | usb | vxi

Type of VISA resource, returned as one of the supported VISA interfaces. Some properties and object functions are specific to an interface type.

See Also

visadev

Introduced in R2021a

visastatus

Check status of VISA resource

Syntax

```
tf = visastatus(v)
[tf,status] = visastatus(v)
```

Description

`tf = visastatus(v)` returns `1` (`true`) if the VISA resource `v` has requested and is ready for service and returns `0` (`false`) if it has not requested service.

`[tf,status] = visastatus(v)` returns the full status byte register containing event information.

Examples

Check VISA Resource Status

Create a connection to an oscilloscope using the VISA-GPIB interface.

```
v = visadev("GPIB0::11::INSTR");
```

Configure the oscilloscope to request service when a command error occurs.

```
writeline(v,"*CLS");
writeline(v,"*PSC 0");
writeline(v,"*ESE 0");
writeline(v,"DESE 0");
writeline(v,"*SRE 32");
```

Send the `"Volt?"` query to the oscilloscope. Since it is an invalid command, a command error occurs.

```
writeline(v,"Volt?");
```

Check whether the oscilloscope has requested service.

```
visastatus(v)
```

```
ans =

  logical

   1
```

Since a command error occurs when you send the "Volt?" command, the oscilloscope has requested service.

Input Arguments

v — VISA resource
visadev object

VISA resource, specified as a visadev object.

Example: visastatus(v) checks the status of the VISA resource service request.

Output Arguments

tf — True or false result
1 | 0

True or false result, returned as a 1 or 0 of data type logical.

Data Types: logical

status — Status byte register
double | 0 to 255

Status byte register, returned as a positive integer from 0 to 255. You can convert this value to its binary value using dec2bin and interpret each bit as information about the VISA resource.

Each bit is associated with a type of event. For example, bit 6 (RQS) indicates the status of the VISA resource service request.

Status Byte Register Bits

Bit	Label	Description
0-3	–	Instrument-specific summary messages.
4	MAV	The Message Available bit indicates if data is available in the Output Queue. MAV is 1 if the Output Queue contains data. MAV is 0 if the Output Queue is empty.
5	ESB	The Event Status bit indicates if one or more enabled events have occurred. ESB is 1 if an enabled event occurs. ESB is 0 if no enabled events occur. You enable events with the Standard Event Status Enable Register.
6	MSS	The Master Summary Status summarizes the ESB and MAV bits. MSS is 1 if either MAV or ESB is 1. MSS is 0 if both MAV and ESB are 0. This bit is obtained from the *STB? command.
	RQS	The Request Service bit indicates that the instrument requests service. This bit can be used for serial polling.
7	–	Instrument-specific summary message.

Data Types: double

See Also
visadev

Topics
"Execute Serial Polls" on page 5-29

Introduced in R2021a

visatrigger

Send trigger message to GPIB or VXI instruments

Syntax

```
visatrigger(v)
```

Description

`visatrigger(v)` sends a trigger message to all GPIB or VXI instruments on the same bus as `v`. This function is only for VISA-GPIB and VISA-VXI interfaces.

Examples

Send Trigger Message to VISA-GPIB Instruments

Create a connection to an instrument associated with bus `GPIB0` using the VISA-GPIB interface.

```
v = visadev("GPIB0::1::0::INSTR");
```

Send a trigger command to all GPIB instruments on `GPIB0`.

```
visatrigger(v);
```

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `visatrigger(v)` sends a trigger message.

See Also

`visadev`

Topics
"Send Trigger Message" on page 5-27

Introduced in R2021a

write

Write data to serial port

Syntax

```
write(device,data,datatype)
```

Description

`write(device,data,datatype)` writes the row or column vector `data` to the specified serial port connection `device`. The function writes the data in the specified data type `datatype`, regardless of the format in `data`. The function suspends MATLAB execution until the specified values are written.

Examples

Write and Read Data with Serial Port Device

Create a connection to a serial port device. In this example, the serial port at `COM3` is connected to a loopback device.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Write the values `[1,2,3,4,5]` in uint8 format.

```
write(device,1:5,"uint8")
```

Since the port is connected to a loopback device, the data you write to the device is returned to MATLAB. Read all the data.

```
read(device,5,"uint8")
```

```
ans = 1×5

     1     2     3     4     5
```

Input Arguments

device — Serial port connection
`serialport` object

Serial port connection, specified as a `serialport` object.

Example: `write(device,1:5,"uint8")` writes to the serial port connection `device`.

`data` — Numeric or ASCII data
numeric | character vector | string scalar

Numeric or ASCII data, specified as a row (1-by-`N`) or column (`N`-by-1) vector of numeric values or as a character vector or string scalar of text. For all numeric `datatype` types, `data` is a row vector of values.

Example: `write(device,[20:24],"int16")` writes the values `[20,21,22,23,24]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`datatype` — Size and format of each value
`"uint8"` | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to write for each value and the interpretation of those bytes as a MATLAB data type. For ASCII text, you can specify `datatype` as either `"char"` or `"string"`.

Example: `write(device,1:5,"int16")` writes data as int16 data type.

Data Types: `char` | `string`

See Also

Functions
`serialport` | `read` | `writeline` | `writebinblock`

Introduced in R2019b

write

Write data to remote host over TCP/IP

Syntax

```
write(t,data)
write(t,data,datatype)
```

Description

`write(t,data)` sends the N-dimensional matrix, `data`, to the remote host specified by the TCP/IP client `t`. The function suspends MATLAB execution until the specified values are written to the remote host.

`write(t,data,datatype)` sends `data` in the form specified by `datatype`, regardless of the format in the matrix of data.

Examples

Write and Read uint8 Data from Remote Host

Create a TCP/IP client connection called `t`, connecting to a TCP/IP echo server with port 4000. To do so, you must have an `echotcpip` server running on port 4000.

```
echotcpip("on",4000)
t = tcpclient("localhost",4000)
```

```
t = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

The `write` function synchronously writes data to the remote host connected to `t`. First specify the data and then write the data. The function suspends MATLAB execution until the specified number of values is written to the remote host.

Assign 10 bytes of `uint8` data to the variable `data`.

```
data = uint8(1:10)
```

```
data = 1×10 uint8 row vector

    1    2    3    4    5    6    7    8    9   10
```

View the data.

```
whos data
```

```
  Name      Size            Bytes  Class    Attributes

  data      1x10               10  uint8
```

Write data to the echo server.

```
write(t,data)
```

Confirm the success of the writing operation by viewing the `NumBytesAvailable` property.

```
t.NumBytesAvailable
```

```
ans = 10
```

Since the client is connected to an echo server, the data you write to the server is returned to the client. Read all the bytes of data available.

```
read(t)
```

```
ans = 1×10 uint8 row vector

    1    2    3    4    5    6    7    8    9   10
```

Using the `read` function with no arguments reads all available bytes of data from `t` connected to the remote host and returns the data. The number of values read is determined by the `NumBytesAvailable` property, which is the number of bytes available in the input buffer.

Close the connection between the TCP/IP client and the remote host by clearing the object. Turn off the `echotcpip` server.

```
clear t
echotcpip("off")
```

Input Arguments

t — TCP/IP client
`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `write(t,data)` writes to the TCP/IP client `t`.

data — Numeric or ASCII data
numeric | character vector | string scalar

Numeric or ASCII data, specified as a row (1-by-N) or column (N-by-1) vector of numeric values or as a character vector or string scalar of text. For all numeric `datatype` types, `data` is a row vector of values.

Example: `write(t,1:5)` writes the values `[1,2,3,4,5]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`datatype` — Size and format of each value
`"uint8"` (default) | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to write for each value and the interpretation of those bytes as a MATLAB data type.

Example: `write(t,1:5,"double")` writes the values `[1,2,3,4,5]` as double data.

Data Types: `char` | `string`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`tcpclient` | `read` | `writeline`

Topics
"Create TCP/IP Client and Configure Settings" on page 7-3
"Write and Read Data over TCP/IP Interface" on page 7-7

Introduced in R2014b

write

Write data from TCP/IP server

Syntax

```
write(t,data)
write(t,data,datatype)
```

Description

`write(t,data)` writes the row or column vector `data` from the TCP/IP server `t` to the client connected to it. The value of the `Connected` property of `t` must be `1` (`true`) before you can write from it.

`write(t,data,datatype)` writes `data` in the form specified by `datatype`, regardless of the format in `data`.

Examples

Write `uint8` Data from TCP/IP Server

Create a TCP/IP server that listens for a client connection request at the specified port and IP address. Then, write data from the server to the connected client.

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server =
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client =
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
```

```
    NumBytesAvailable: 0

  Show all properties, functions
```

Display the values of the `Connected`, `ClientAddress`, and `ClientPort` properties for `server`.

```
server
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 1
        ClientAddress: "127.0.0.1"
           ClientPort: 49653
    NumBytesAvailable: 0

  Show all properties, functions
```

The output shows that `server` successfully accepts a request from `client` and that `client` establishes a connection to `server`.

Send data to the client by writing it using the `server` object. Since the client is connected to the server, this data is available in the client. Read the data from the `client` object.

```
write(server,[6,9,14,26,27,42],"uint8")
read(client,client.NumBytesAvailable)
```

```
ans = 1×6 uint8 row vector

    6    9   14   26   27   42
```

Write String Data from TCP/IP Server

Create a TCP/IP server that listens for a client connection request at the specified port and IP address. Then, write data from the server to the connected client.

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0
```

```
  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

See the values of the `Connected`, `ClientAddress`, and `ClientPort` properties for `server`.

```
server
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 1
        ClientAddress: "127.0.0.1"
           ClientPort: 65136
    NumBytesAvailable: 0

  Show all properties, functions
```

The output shows that `server` successfully accepts a request from `client` and that `client` establishes a connection to `server`.

Send data to the client by writing it using the `server` object. Since the client is connected to the server, this data is available in the client. Read this data from the `client` object.

```
write(server,"hello world","string")
read(client,11,"string")
```

```
ans = 
"hello world"
```

Input Arguments

t — TCP/IP server
`tcpserver` object

TCP/IP server, specified as a `tcpserver` object.

Example: `write(t,10)` writes from the TCP/IP server `t` to the client connected to it.

`data` — Numeric or ASCII data
numeric | character vector | string scalar

Numeric or ASCII data, specified as a row (1-by-N) or column (N-by-1) vector of numeric values or as a character vector or string scalar of text. For all numeric `datatype` types, `data` is a row vector of values.

Example: `write(t,300)` writes the value `300`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`datatype` — Size and format of each value
`"uint8"` (default) | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to write for each value and the interpretation of those bytes as a MATLAB data type.

For any read or write operation, the data type is converted to `uint8` for the data transfer. After the transfer, the data type reverts to the specified `datatype`.

Example: `write(t,1:5,"double")` writes the values `[1,2,3,4,5]` as double data.

Data Types: `char` | `string`

See Also
`tcpserver` | `read` | `writeline`

Introduced in R2021a

write

Write data to UDP socket

Syntax

```
write(u,data,destinationAddress,destinationPort)
write(u,data)
write(u,data,datatype)
write(u,data,datatype,destinationAddress,destinationPort)
```

Description

`write(u,data,destinationAddress,destinationPort)` sends the vector of values in `data` to the specified IP `destinationAddress` and `destinationPort`, using the default `datatype` precision of `uint8`. The function waits until the requested number of values are written to the UDP socket, or until a timeout occurs.

`write(u,data)` sends the vector of values in `data` to the last used `destinationAddress` and `destinationPort`, using the default `datatype` precision of `uint8`. If you do not specify `destinationAddress` and `destinationPort` in a previous call to `write` or `writeline` for the UDP socket `u`, this syntax throws an error.

`write(u,data,datatype)` sends the vector of values in `data` using the specified `datatype` precision, regardless of the actual type of `data`.

`write(u,data,datatype,destinationAddress,destinationPort)` sends the vector of values in `data` using the specified `datatype`, `destinationAddress`, and `destinationPort`.

Examples

Write Values to UDP Socket

Write a vector of unsigned 8-bit values to a UDP socket.

```
u = udpport;
write(u, 1:5, "uint8", "192.1.5.15", 20);
```

For future writes to the same destination address and port for the `udpport` object `u`, you can omit the `destinationAddress` and `destinationPort` arguments.

Write a vector of single values.

```
write(u,1:10,"single");
```

Input Arguments

u — UDP socket
`udpport` object

UDP socket, specified as a `udpport` object.

Example: `u = udpport`

Data Types: `udpport object`

`data` — Vector of values to write
numeric | string | character vector

Vector of values to write, specified as a row (1-by-`N`) or column (`N`-by-1) array of numeric data, string, or character vector. If the size of `data` is greater than the `OutputDatagramSize` property of the UDP socket object `u`, the function splits the data into multiple packets.

Example: `"Hello world"`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

`destinationAddress` — Destination address to write to
string | character vector

Destination address to write to, specified as a string or character vector. If you do not provide this value, the packet is sent to the last used `destinationAddress`. When you write to this address and port for the first time, `destinationAddress` is required.

Example: `"192.1.5.15"`

Data Types: `char` | `string`

`destinationPort` — Destination port to write to
numeric

Destination port to write to, specified as a numeric value from 0 to 65535. If you do not provide this argument and the `destinationAddress`, the function writes to the last used `destinationPort`. When you write to this address and port for the first time, `destinationPort` is required.

Example: `5110`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`datatype` — MATLAB data type for each value
`"uint8"` (default) | string | character vector

MATLAB data type for each value, specified as a string or character vector. `datatype` specifies the number of bits written for each value, and the interpretation of those bits as a MATLAB data type. Allowed values are `"int8"`, `"int16"`, `"int32"`, `"int64"`, `"uint8"`, `"uint16"`, `"uint32"`, `"uint64"`, `"double"`, `"single"`, `"char"`, and `"string"`.

Example: `"uint16"`

Data Types: `char` | `string`

See Also

Functions
`udpport` | `read` | `readline` | `writeline`

Introduced in R2020b

write

Write data to VISA resource

Syntax

```
write(v,data)
write(v,data,datatype)
```

Description

`write(v,data)` writes the row or column vector `data` to the VISA resource `v`.

`write(v,data,datatype)` writes `data` in the form specified by `datatype`, regardless of the format in `data`.

Examples

Write `uint8` Data to VISA Resource

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Write the values `[1,2,3,4,5]` as `uint8` data to the VISA resource `v`.

```
write(v,1:5)
```

Write String Data to VISA Resource

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Write a string of data to the VISA resource `v`.

```
write(v,"Hello, world!","string")
```

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `write(v,10)` writes to the VISA resource `v`.

data — Numeric or ASCII data
numeric | character vector | string scalar

Numeric or ASCII data, specified as a row (1-by-N) or column (N-by-1) vector of numeric values or as a character vector or string scalar of text. For all numeric `datatype` types, `data` is a row vector of values.

Example: `write(v,300)` writes the value `300`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

datatype — Size and format of each value
`"uint8"` (default) | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to write for each value and the interpretation of those bytes as a MATLAB data type.

For any read or write operation, the data type is converted to `uint8` for the data transfer. After the transfer, the data type reverts to the specified `datatype`.

Example: `write(v,1:5,"double")` writes the values `[1,2,3,4,5]` as double data.

Data Types: `char` | `string`

See Also
`visadev` | `read` | `writeline`

Introduced in R2021a

write

Write binary data to SPI instrument

Syntax

```
write(OBJ, A)
```

Description

`write(OBJ, A)` writes the data, A, to the SPI instrument connected to interface object, OBJ. OBJ must be a 1-by-1 SPI interface object. By default the `'uint8'` precision is used.

The interface object must be connected to the device with the `connect` function before any data can be read from or written to the device, otherwise an error is returned. A connected interface object has a `ConnectionStatus` property value of connected.

The SPI protocol operates in full duplex mode, input and output data transfers happen simultaneously. For every byte written to the device, a byte is read back from the device. This function will automatically flush the incoming data.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Examples

This example shows how to create a SPI object `s`, and read and write data.

Construct a `spi` object called `s` using `Vendor 'aardvark'`, with `BoardIndex` of `0`, and `Port` of `0`.

```
s = spi('aardvark', 0, 0);
```

Connect to the chip.

```
connect(s);
```

Write to the chip.

```
dataToWrite = [2 0 0 255]
write(s, dataToWrite);
```

Disconnect the SPI device and clean up by clearing the object.

```
disconnect(s);
clear('s');
```

Introduced in R2013b

write

Perform a write operation to the connected MODBUS server

Syntax

```
write(m,target,address,values)
write(m,target,address,values,serverId,'precision')
```

Description

`write(m,target,address,values)` writes data to MODBUS object `m` to target type `target` at the starting address `address` using the values to read `values`. You can write to coils or holding registers.

`write(m,target,address,values,serverId,'precision')` additionally specifies `serverId`, which is the address of the server to send the read command to, and the `precision`, which is the data format of the register being read.

`serverId` can be used for both coils and holding registers, and `precision` can be used for registers only. You can use either argument by itself, or use both arguments together when the write target is holding registers.

Examples

Write Coils Over MODBUS

If the write target is coils, the function writes a contiguous sequence of 1–1968 coils to either on or off in a remote device. A coil is a single output bit. A value of `1` indicates the coil is on and a value of `0` means it is off.

Write to 4 coils, starting at address 8289. The `address` parameter is the starting address of the coils to write to, and it is a double. The `values` parameter is an array of values to write.

```
write(m,'coils',8289,[1 1 0 1])
```

You can also create a variable for the values to write.

```
values = [1 1 0 1];
write(m,'coils',8289,values)
```

Write Holding Registers Over MODBUS

If the write target is holding registers, the function writes a block of 1–123 contiguous registers in a remote device. Values whose representation is greater than 16 bits are stored in consecutive register addresses.

Set the register at address 49153 to 2000.

```
write(m,'holdingregs',49153,2000)
```

Specify Server ID and Precision Options for the Write Operation

You can write to coils or holding registers and also specify the optional parameter for server ID, and you can specify precision for holding registers. You can set either option by itself or set both the `serverId` option and the `precision` option together. Both options should be listed after the required arguments.

Write 3 values, starting at address 29473, at Server ID 2, converting to `single` precision.

```
write(m,'holdingregs',29473,[928.1 50.3 24.4],2,'single')
```

Input Arguments

`target` — Target area to write to
character vector | string

Target area to write to, specified as a character vector or string. You can perform a MODBUS write operation on two types of targets: coils and holding registers, so you must set the target type as either `'coils'` or `'holdingregs'`. Target must be the first argument after the object name. This example writes to 4 coils starting at address 8289.

Example: `write(m,'coils',8289,[1 1 0 1])`

Data Types: `char`

`address` — Starting address to write to
double

Starting address to write to, specified as a double. Address must be the second argument after the object name. This example writes to 6 coils starting at address 5200.

Example: `write(m,'coils',5200,[1 1 0 1 1 0])`

Data Types: `double`

`values` — Array of values to write
double | array of doubles

Array of values to write, specified as a double or array of doubles. `values` must be the third argument after the object name. If the target is coils, valid values are `0` and `1`. If the target is holding registers, valid values must be in the range of the specified precision. You can include the array of values in the syntax, as shown here, or use a variable for the values.

This example writes to 4 coils starting at address 8289.

Example: `write(m,'coils',8289,[0 1 0 1])`

Data Types: `double`

`serverId` — Address of the server to send the write command to
double

Address of the server to send the write command to, specified as a double. Server ID must be specified after the object name, target, address, and values. If you do not specify a `serverId`, the

default of `1` is used. Valid values are `0-247`, with `0` being the broadcast address. This example writes 8 coils starting at address 1 from server ID 3.

Example: `write(m,'coils',1,[1 1 1 1 0 0 0 0],3);`

Data Types: `double`

`precision` — Data format of the register being written to on the MODBUS server
character vector | string

Data format of the register being written to on the MODBUS server, specified as a character vector or string. Precision must be specified after the object name, target, address, and values. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

Note that `precision` does not refer to the return type, which is always `'double'`. It specifies how to interpret the register data.

This example writes to 4 holding registers starting at address 2 using a precision of `'uint32'`.

Example: `write(m,'holdingregs',2,[100 200 300 500],'uint32');`

Data Types: `char`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`modbus` | `read` | `writeRead` | `maskWrite`

Topics

“Create a MODBUS Connection” on page 11-3
“Configure Properties for MODBUS Communication” on page 11-5
“Write Data to a MODBUS Server” on page 11-14

Introduced in R2017a

writeAndRead

Write and read binary data from SPI instrument

Syntax

```
A = writeAndRead(OBJ, dataToWrite)
```

Description

`A = writeAndRead(OBJ, dataToWrite)` writes the data, `dataToWrite`, to the instrument connected to interface object OBJ and reads the data available from the instrument as a result of writing `dataToWrite`. OBJ must be a 1-by-1 SPI interface object. Values are written and read as `uint8` data.

The interface object must be connected to the device using the `connect` function before any data can be read from the device, otherwise an error is returned. A connected interface object has a `ConnectionStatus` property value of `connected`.

SPI protocol operates in full duplex mode, so input and output data transfers happen simultaneously. For every byte written to the device, a byte is read back from the device.

For more information on using the SPI interface and this function, see "Configuring SPI Communication" on page 10-3 and "Transmitting Data Over the SPI Interface" on page 10-7.

Note To get a list of options you can use on a function, press the **Tab** key after entering a function on the MATLAB command line. The list expands, and you can scroll to choose a property or value. For information about using this advanced tab completion feature, see "Using Tab Completion for Functions" on page 2-4.

Examples

This example shows how to create a SPI object `s`, and read and write data.

Construct a `spi` object called `s` using `Vendor 'aardvark'`, with `BoardIndex` of `0`, and `Port` of `0`.

```
s = spi('aardvark', 0, 0);
```

Connect to the chip.

```
connect(s);
```

Read and write to the chip.

```
dataToWrite = [2 0 0 255]
data = writeAndRead(s, dataToWrite);
```

Disconnect the SPI device and clean up by clearing the object.

```
disconnect(s);
clear('s');
```

Introduced in R2013b

writebinblock

Write one binblock of data to serial port

Syntax

```
writebinblock(device,data,precision)
```

Description

`writebinblock(device,data,precision)` converts `data` into a binblock and writes it to the serial port. `data` can be a string, character vector, or numeric 1-by-N matrix. The written data has the specified precision regardless of the format in the matrix. The function blocks MATLAB and waits until the binblock data is written to the serial port.

Examples

Write Binblock of uint8 Data

Convert `[1,2,3,4,5]` to a binblock and write it to the serial port as uint8.

```
s = serialport("COM3",9600);
  ⋮
writebinblock(s,1:5,"uint8")
```

Input Arguments

device — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

data — Numeric or ASCII data
1-by-N numeric array | string | char

Numeric or ASCII data to write to serial port, specified as a string, character vector, or 1-by-N vector of numeric values. For all numeric `precision` types, `data` is a row vector of values.

Example: `[20:24]`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

precision — Size and format of each value
`'uint8'` | `'int8'` | `'uint16'` | `'int16'` | `'uint32'` | `'int32'` | `'uint64'` | `'int64'` | `'single'` | `'double'` | `'char'` | `'string'`

Size and format of each value, specified as a character vector or string. `precision` determines the number of bits to write for each value and its format as a MATLAB data type. For ASCII text, you can specify `precision` as either `'char'` or `'string'`.

Example: `'int16'`

Data Types: `char` | `string`

See Also

Functions
`serialport` | `write` | `writeline` | `readbinblock`

Introduced in R2019b

writebinblock

Write one binblock of data to remote host over TCP/IP

Syntax

```
writebinblock(t,data,datatype)
```

Description

`writebinblock(t,data,datatype)` writes a binblock of `data` in the form specified by `datatype` to the remote host specified by the TCP/IP client `t`. The function suspends MATLAB execution until the specified values are written to the remote host.

Examples

Write and Read Binblock of Data from Remote Host

Create a TCP/IP client connection called `t`, connecting to a TCP/IP echo server with port 4000. To do so, you must have an `echotcpip` server running on port 4000.

```
echotcpip("on",4000)
t = tcpclient("localhost",4000)
```

```
t =
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Write the values `[1,2,3,4,5]` as a binblock in uint8 format.

```
writebinblock(t,1:5,"uint8")
```

Write another binblock of data. Write the values `[6,7,8,9,10]` as double data.

```
writebinblock(t,6:10,"double")
```

Since the client is connected to an echo server, the data you write to the server is returned to the client. Read the first binblock of data that you wrote.

```
readbinblock(t)
```

```
ans = 1×5

     1     2     3     4     5
```

Read a binblock of data again to return the second set of values that you wrote. Specify the data as `double`.

```
readbinblock(t,"double")
```

```
ans = 1×5

     6     7     8     9    10
```

Close the connection between the TCP/IP client and the remote host by clearing the object. Turn off the `echotcpip` server.

```
clear t
echotcpip("off")
```

Input Arguments

t — TCP/IP client
`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `writebinblock(t,1:5,"uint8")` writes a binblock of data to the TCP/IP client `t`.

data — Numeric or ASCII data
1-by-N numeric array | character vector | string scalar

Numeric or ASCII data, specified as a 1-by-N vector of numeric values or as a character vector or string scalar of text. For all numeric `datatype` types, `data` is a row vector of values.

Example: `writebinblock(t,1:5,"uint8")` writes the values `[1,2,3,4,5]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

datatype — Size and format of each value
`"uint8"` | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to write for each value and the interpretation of those bytes as a MATLAB data type.

Example: `writebinblock(t,1:5,"double")` writes the values `[1,2,3,4,5]` as double data.

Data Types: `char` | `string`

See Also

`tcpclient` | `write` | `writeline` | `readbinblock`

Introduced in R2020b

writebinblock

Write one binblock of data from TCP/IP server

Syntax

`writebinblock(t,data,datatype)`

Description

`writebinblock(t,data,datatype)` writes a binblock of `data` in the form specified by `datatype` from the TCP/IP server `t` to the client connected to it. The value of the `Connected` property of `t` must be `1` (`true`) before you can write from it. The function suspends MATLAB execution until the specified values are written to the remote host.

Examples

Write Binblock of Data from TCP/IP Server

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Write the values `[1,2,3,4,5]` from the server to the client by writing it to the `server` object as a binblock in `uint8` format.

```
writebinblock(server,1:5,"uint8")
```

Write another binblock of data. Write the values `[6,7,8,9,10]` as double data.

```
writebinblock(server,6:10,"double")
```

Since the client is connected to the server, the data you write to the server is available to be read from the `client` object. Read the first binblock of data.

```
readbinblock(client)
```

```
ans = 1×5

     1     2     3     4     5
```

Read a binblock of data again to return the second set of values. Specify the data as `double`.

```
readbinblock(client,"double")
```

```
ans = 1×5

     6     7     8     9    10
```

Input Arguments

t — TCP/IP server
`tcpserver` object

TCP/IP server, specified as a `tcpserver` object.

Example: `writebinblock(t,1:5,"uint8")` writes a binblock of data to the TCP/IP client connected to the server `t`.

data — Numeric or ASCII data
1-by-N numeric array | character vector | string scalar

Numeric or ASCII data, specified as a 1-by-N vector of numeric values or as a character vector or string scalar of text. For all numeric `datatype` types, `data` is a row vector of values.

Example: `writebinblock(t,1:5,"uint8")` writes the values `[1,2,3,4,5]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

datatype — Size and format of each value
`"uint8"` | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to write for each value and the interpretation of those bytes as a MATLAB data type.

Example: `writebinblock(t,1:5,"double")` writes the values `[1,2,3,4,5]` as double data.

Data Types: `char` | `string`

See Also

`tcpserver` | `write` | `writeline` | `readbinblock`

Introduced in R2021a

writebinblock

Write one binblock of data to VISA resource

Syntax

```
writebinblock(v,data,datatype)
```

Description

`writebinblock(v,data,datatype)` writes a binblock of `data` in the form specified by `datatype` to the VISA resource `v`. The function suspends MATLAB execution until the specified values are written to the remote host.

Examples

Write Binblock of Data to VISA Resource

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Write the values `[1,2,3,4,5]` as a binblock in `double` format to the VISA resource `v`.

```
writebinblock(v,1:5,"double")
```

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `writebinblock(v,1:5,"uint8")` writes a binblock of data to the VISA resource `v`.

data — Numeric or ASCII data
1-by-N numeric array | character vector | string scalar

Numeric or ASCII data, specified as a 1-by-N vector of numeric values or as a character vector or string scalar of text. For all numeric `datatype` types, `data` is a row vector of values.

Example: `writebinblock(v,1:5,"uint8")` writes the values `[1,2,3,4,5]`.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `char` | `string`

datatype — Size and format of each value
`"uint8"` | `"int8"` | `"uint16"` | `"int16"` | `"uint32"` | `"int32"` | `"uint64"` | `"int64"` | `"single"` | `"double"` | `"char"` | `"string"`

Size and format of each value, specified as a character vector or string. `datatype` determines the number of bytes to write for each value and the interpretation of those bytes as a MATLAB data type.

Example: `writebinblock(v,1:5,"double")` writes the values `[1,2,3,4,5]` as double data.

Data Types: `char` | `string`

See Also

`visadev` | `write` | `writeline` | `readbinblock`

Introduced in R2021a

writeline

Write line of ASCII data to serial port

Syntax

```
writeline(device,data)
```

Description

`writeline(device,data)` writes the ASCII text `data` followed by the terminator to the specified serial port. The function suspends MATLAB execution until the data and terminator are written.

Examples

Write and Read Line of ASCII Data from Serial Port Device

Create a connection to a serial port device. In this example, the serial port at `COM3` is connected to a loopback device.

```
device = serialport("COM3",9600)
```

```
device = 

  Serialport with properties:

                 Port: "COM3"
             BaudRate: 9600
    NumBytesAvailable: 0

  Show all properties, functions
```

Check the default ASCII terminator.

```
device.Terminator
```

```
ans = 

    "LF"
```

Set the terminator to `"CR"` and write a string of ASCII data. The `writeline` function automatically appends the terminator to the data.

```
configureTerminator(device,"CR")
writeline(device,"hello")
```

Write another string of ASCII data with the terminator automatically appended.

```
writeline(device,"world")
```

Since the port is connected to a loopback device, the data you write to the device is returned to MATLAB. Read a string of ASCII data. The `readline` function returns data until it reaches a terminator.

```
readline(device)
```

```
ans = 

    "hello"
```

Read a string of ASCII data again to return the second string that you wrote.

```
readline(device)
```

```
ans = 

    "world"
```

Clear the serial port connection.

```
clear device
```

Input Arguments

device — Serial port connection
`serialport` object

Serial port connection, specified as a `serialport` object.

Example: `writeline(device,1:5)` writes to the serial port connection `device`.

data — ASCII data
character vector | string scalar

ASCII data to write, specified as a character vector or string scalar of text.

Example: `writeline(device,"*IDN?")` writes the ASCII string `"*IDN?"`

Data Types: `char` | `string`

See Also

Functions
`serialport` | `readline` | `write` | `configureTerminator` | `writebinblock` | `writeread`

Introduced in R2019b

writeline

Write line of ASCII data to remote host over TCP/IP

Syntax

```
writeline(t,data)
```

Description

`writeline(t,data)` writes the ASCII text `data` followed by the terminator to the remote host specified by the TCP/IP client `t`. The function suspends MATLAB execution until the data and terminator are written.

Examples

Write and Read Line of ASCII Data from Remote Host

Create a TCP/IP client connection called `t`, connecting to a TCP/IP echo server with port 4000. To do so, you must have an `echotcpip` server running on port 4000.

```
echotcpip("on",4000)
t = tcpclient("localhost",4000)
```

```
t =
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Check the default ASCII terminator.

```
t.Terminator
```

```
ans =
"LF"
```

Set the terminator to `"CR"` and write a string of ASCII data. The `writeline` function automatically appends the terminator to the data.

```
configureTerminator(t,"CR")
writeline(t,"hello")
```

Write another string of ASCII data with the terminator automatically appended.

```
writeline(t,"world")
```

Since the client is connected to an echo server, the data you write to the server is returned to the client. Read a string of ASCII data. The `readline` function returns data until it reaches a terminator.

```
readline(t)
```

```
ans = 
"hello"
```

Read a string of ASCII data again to return the second string that you wrote.

```
readline(t)
```

```
ans = 
"world"
```

Close the echo server and clear the TCP/IP client connection.

```
echotcpip("off")
clear t
```

Input Arguments

t — TCP/IP client
`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `writeline(t,data)` writes ASCII data to the TCP/IP client `t`.

data — ASCII data
character vector | string scalar

ASCII data to write, specified as a character vector or string scalar of text.

Example: `writeline(t,"helloworld")` writes the ASCII data `"helloworld"`.

Data Types: `char` | `string`

See Also

Functions
`tcpclient` | `configureTerminator` | `readline` | `write`

Introduced in R2020b

writeline

Write line of ASCII data from TCP/IP server

Syntax

```
writeline(t,data)
```

Description

`writeline(t,data)` writes the ASCII text `data` followed by the terminator from the TCP/IP server `t` to the client connected to it. The value of the `Connected` property of `t` must be `1` (`true`) before you can write from it. The function suspends MATLAB execution until the data and terminator are written.

Examples

Write Line of ASCII Data from TCP/IP Server

Create a TCP/IP server that listens for connections at `localhost` and port 4000.

```
server = tcpserver("localhost",4000)
```

```
server = 
  TCPServer with properties:

        ServerAddress: "127.0.0.1"
           ServerPort: 4000
            Connected: 0
        ClientAddress: ""
           ClientPort: []
    NumBytesAvailable: 0

  Show all properties, functions
```

Create a TCP/IP client to connect to your server object using `tcpclient`. You must specify the same IP address and port number you use to create `server`.

```
client = tcpclient("localhost",4000)
```

```
client = 
  tcpclient with properties:

              Address: 'localhost'
                 Port: 4000
    NumBytesAvailable: 0

  Show all properties, functions
```

Check the default ASCII terminator for the server.

```
server.Terminator

ans = 
"LF"
```

Set the terminators for both the server and client to `"CR"`. The TCP/IP server and its connected client must have the same terminator.

```
configureTerminator(server,"CR")
configureTerminator(client,"CR")
```

Write a string of ASCII data from the server to the client by writing it to the `server` object. The `writeline` function automatically appends the terminator to the data.

```
writeline(server,"hello")
```

Write another string of ASCII data with the terminator automatically appended.

```
writeline(server,"world")
```

Since the client is connected to the server, the data you write is available in the client. Read a string of ASCII data from the `client` object. The `readline` function returns data until it reaches a terminator.

```
readline(client)

ans = 
"hello"
```

Read a string of ASCII data again to return the second string.

```
readline(client)

ans = 
"world"
```

Input Arguments

t — TCP/IP server
`tcpserver` object

TCP/IP server, specified as a `tcpserver` object.

Example: `writeline(t,data)` writes ASCII data to the TCP/IP client connected to the server `t`.

data — ASCII data
character vector | string scalar

ASCII data to write, specified as a character vector or string scalar of text.

Example: `writeline(t,"helloworld")` writes the ASCII data `"helloworld"`.

Data Types: `char` | `string`

See Also
`tcpserver` | `configureTerminator` | `readline` | `write`

Introduced in R2021a

writeline

Write line of ASCII data to UDP socket

Syntax

```
writeline(u,data,destinationAddress,destinationPort)
writeline(u,data)
```

Description

`writeline(u,data,destinationAddress,destinationPort)` writes the ASCII data `data`, followed by the terminator, to the specified IP `destinationAddress` and `destinationPort`. `u` must be a byte-type `udpport` object. The function waits until the data and terminator are written to the UDP socket, or until a timeout occurs.

`writeline(u,data)` writes the ASCII data to the last used `destinationAddress` and `destinationPort`. If you do not specify `destinationAddress` and `destinationPort` in a previous call to `write` or `writeline` for the UDP socket `u`, this syntax throws an error.

Examples

Write ASCII Line to UDP Socket

Write start and stop commands to a UDP socket.

Create a UDP socket and write an ASCII `"START"` string to a specified address and port.

```
u = udpport;
writeline(u,"START","192.1.5.15",20)
```

The function appends the default terminator to the written string.

Write an ASCII `"STOP"` string to the same address and port.

```
writeline(u,"STOP")
```

Input Arguments

u — UDP socket
`udpport` object

UDP socket, specified as a `udpport` object.

Example: `u = udpport`

Data Types: `udpport object`

`data` — String to write to `udpport` socket
string | character vector

String to write to `udpport` socket, specified as a string or character vector. The terminator is appended. If the size of `data` is greater than the `OutputDatagramSize` property of the UDP socket object `u`, the function splits the data into multiple packets.

Example: `"Hello world"`

Data Types: `char` | `string`

`destinationAddress` — Destination address to write to
string | character vector

Destination address to write to, specified as a string or character vector. If you do not provide this argument, the function writes to the last used `destinationAddress`. When you write to this address and port for the first time, `destinationAddress` is required.

Example: `"192.1.5.15"`

Data Types: `char` | `string`

`destinationPort` — Destination port to write to
numeric

Destination port to write to, specified as a numeric value from 0 to 65535. If you do not provide this argument and the `destinationAddress`, the function writes to the last used `destinationPort`. When you write to this address and port for the first time, `destinationPort` is required.

Example: `5110`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

See Also

Functions
`udpport` | `readline` | `configureTerminator`

Introduced in R2020b

writeline

Write line of ASCII data to VISA resource

Syntax

```
writeline(v,data)
```

Description

`writeline(v,data)` writes the ASCII text `data` followed by the terminator to the VISA resource `v`. The function suspends MATLAB execution until the data and terminator are written.

Examples

Write ASCII Line to VISA Resource

Create a connection to a VISA resource. This example shows a connection to a device with the alias `COM4` using the VISA-Serial interface.

```
v = visadev("COM4");
```

Set the terminator to `"CR/LF"` and write a string of ASCII data to the VISA resource `v`. The `writeline` function automatically appends the terminator to the data.

```
configureTerminator(v,"CR/LF")
writeline(v,"START")
```

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `writeline(v,data)` writes ASCII data to the VISA resource `v`.

data — ASCII data
character vector | string scalar

ASCII data to write, specified as a character vector or string scalar of text.

Example: `writeline(v,"helloworld")` writes the ASCII data `"helloworld"`.

Data Types: `char` | `string`

See Also

`visadev` | `configureTerminator` | `readline` | `write`

Introduced in R2021a

writeread

Write command to serial port and read response

Syntax

```
response = writeread(device,command)
```

Description

`response = writeread(device,command)` writes the ASCII text `command` followed by the terminator to the specified serial port device, then reads the ASCII text returned from the device and assigns it to `response`. You can use this function to query an instrument connected to the serial port. The function blocks MATLAB and waits until the `command` and `response` are complete, or a timeout occurs.

Examples

Query Serial Device for ID

Write an instrument identification query to a serial instrument and read the response. `writeline` automatically includes the defined terminator.

```
s = serialport("COM3",9600);
configureTerminator(s,"CR")
  ⋮
resp = writeread(s,"*IDN?");
```

Input Arguments

device — Serial port
`serialport` object

Serial port, specified as a `serialport` object.

Example: `serialport()`

command — ASCII text command to device
string | char

ASCII text command to device, specified as a string or character vector.

Example: `"IDN?"`

Data Types: `char` | `string`

Output Arguments

response — ASCII text response from device
string

ASCII text response from device, returned as a string. The terminator is not included.

See Also

Functions
`serialport` | `readline` | `configureTerminator` | `writeline`

Introduced in R2019b

writeread

Write command to remote host over TCP/IP and read response

Syntax

```
response = writeread(t,command)
```

Description

`response = writeread(t,command)` writes the ASCII text `command` followed by the terminator to the remote host specified by the TCP/IP client `t`, then reads the ASCII text returned from the remote host. You can use this function to query the remote host. The function suspends MATLAB execution until the specified `command` is sent to the remote host and a response is received.

Examples

Query a TCP/IP Instrument for ID

Create a TCP/IP client connection to an instrument. In this example, the instrument connected to this network is a Keysight Technologies® (formerly Agilent Technologies®) X-Series Signal Analyzer (N9030A, PXA Signal Analyzer). The specified IP address and port are unique to this example.

```
t = tcpclient("172.31.165.102",5025)
```

```
t = 
  tcpclient with properties:

              Address: '172.31.165.102'
                 Port: 5025
    NumBytesAvailable: 0

  Show all properties, functions
```

Write an instrument identification SCPI command to the instrument and read the response.

```
writeread(t,"*IDN?")
```

```
ans = 
"Agilent Technologies,N9030A,US00071181,A.14.16"
```

The instrument response to the `*IDN?` command identifies the name of the instrument.

Input Arguments

t — TCP/IP client
`tcpclient` object

TCP/IP client, specified as a `tcpclient` object.

Example: `writeread(t,"*IDN?")` sends an ASCII text command to the TCP/IP client `t`.

command — ASCII text command to write to device
character vector | string scalar

ASCII text command to write to device, specified as a character vector or string scalar.

Example: `writeread(t,"*IDN?")` sends the ASCII command `*IDN?`.

Data Types: `char` | `string`

See Also

`tcpclient` | `configureTerminator` | `readline` | `writeline`

Introduced in R2020b

writeread

Write command to VISA resource and read response

Syntax

```
response = writeread(v,command)
```

Description

`response = writeread(v,command)` writes the ASCII text `command` followed by the terminator to the VISA resource `v`, then reads the ASCII text returned from the resource. You can use this function to query the resource. The function suspends MATLAB execution until the specified `command` is sent to the resource and a response is received.

Examples

Query a VISA Resource for ID

Create a connection to a VISA resource. This example shows a connection to a Keysight Technologies® (formerly Agilent Technologies®) X-Series Signal Analyzer (N9030A, PXA Signal Analyzer) using the VISA-TCP/IP interface. The specified resource name is unique to this example.

```
v = visadev("TCPI0::172.31.165.102::inst0::INSTR");
```

Write an instrument identification SCPI command to the instrument and read the response.

```
writeread(v,"*IDN?")
```

```
ans = 
"Agilent Technologies,N9030A,US00071181,A.14.16"
```

The instrument response to the `*IDN?` command identifies the name, model, and serial number of the instrument.

Input Arguments

v — VISA resource
`visadev` object

VISA resource, specified as a `visadev` object.

Example: `writeread(v,"*IDN?")` sends an ASCII text command to the VISA resource `v`.

command — ASCII text command to write to device
character vector | string scalar

ASCII text command to write to device, specified as a character vector or string scalar.

Example: `writeread(v,"*IDN?")` sends the ASCII command `*IDN?`.

Data Types: `char` | `string`

See Also

`visadev` | `configureTerminator` | `readline` | `writeline`

Introduced in R2021a

writeRead

Perform a write then read operation on groups of holding registers in a single MODBUS transaction

Syntax

```
writeRead(m,writeAddress,values,readAddress,readCount)
writeRead(m,writeAddress,values,readAddress,readCount,serverId)
writeRead(m,writeAddress,values,writePrecision,readAddress,readCount,
readPrecision)
```

Description

`writeRead(m,writeAddress,values,readAddress,readCount)` writes data to MODBUS object `m` at the starting address `writeAddress` using the values to write `values`, and then reads data at the starting address `readAddress` using the number of values to read `readCount`.

This function performs a combination of one write operation and one read operation on groups of holding registers in a single MODBUS transaction. The write operation is always performed before the read. The range of addresses to read must be contiguous, and the range of addresses to write must be contiguous, but each is specified independently and may or may not overlap.

`writeRead(m,writeAddress,values,readAddress,readCount,serverId)` additionally uses the `serverId` as the address of the server to send the command to.

`writeRead(m,writeAddress,values,writePrecision,readAddress,readCount, readPrecision)` adds optional precisions for the write and read operations. The `writePrecision` and `readPrecision` arguments specify the data format of the register being read from or written to on the MODBUS server.

Examples

Write and Read Holding Registers

The `writeRead` function is used to perform a combination of one write operation and one read operation on groups of holding registers in a single MODBUS transaction. The write operation is always performed before the read. The range of addresses to read must be contiguous, and the range of addresses to write must be contiguous, but each is specified independently and may or may not overlap.

Write 2 holding registers starting at address 300, and read 4 holding registers starting at address 17250.

```
writeRead(m,300,[500 1000],17250,4)
```

```
ans =

   35647   48923   50873   60892
```

If the operation is successful, it returns an array of doubles, each representing a 16-bit register value, where the first value in the vector corresponds to the register value at the address specified in `readAddress`.

You can optionally create variables for the values to be written, instead of including the array of values in the function syntax. The example could be written this way, using a variable for the values:

```
values = [500 1000];
writeRead(m,300,values,17250,4)
```

```
ans =

   35647   58923   50873   60892
```

Write and Read Holding Registers, and Specify Server ID

Use the `serverId` argument to specify the address of the server to send the command to.

Write 3 holding registers starting at address 400, and read 4 holding registers starting at address 52008 from server ID 6.

```
writeRead(m,400,[1024 512 680],52008,4,6)
```

```
ans =

   38629   24735   29456   39470
```

Write and Read Holding Registers, and Specify Precisions

Use the `writePrecision` and `readPrecision` arguments to specify the data format of the register being read from or written to on the MODBUS server.

Write 3 holding registers starting at address 500, and read 6 holding registers starting at address 52008 from server ID 6. Specify a `writePrecision` of `'uint64'` and a `readPrecision` of `'uint32'`.

```
writeRead(m,500,[1024 512 680],'uint64',52008,6,'uint32',6)
```

```
ans =

   38629   24735   29456   39470   33434   29484
```

Input Arguments

`writeAddress` — Starting address of the registers to write
double

Starting address to write to, specified as a double. `writeAddress` must be the first argument after the object name. This example writes 2 holding registers starting at address 501 and reads 4 holding registers starting at address 11250. The `writeAddress` is `501`.

Example: `writeRead(m,501,[1024 512],11250,4)`

Data Types: `double`

`values` — Array of values to write
double | array of doubles

Array of values to write, specified as a double or array of doubles. Values must be the second argument after the object name. Each value must be in the range `0–65535`. This example writes 2 holding registers starting at address 501 and reads 4 holding registers starting at address 11250. The `values` are `[1024 512]`.

Example: `writeRead(m,501,[1024 512],11250,4)`

Data Types: `double`

`readAddress` — Starting address of the holding registers to read
double

Starting address of the holding registers to read, specified as a double. `readAddress` must be the third argument after the object name. This example writes 2 holding registers starting at address 501 and reads 4 holding registers starting at address 11250. The `readAddress` is `11250`.

Example: `writeRead(m,501,[1024 512],11250,4)`

Data Types: `double`

`readCount` — Number of holding registers to read
double

Number of holding registers to read, specified as a double. `readCount` must be the fourth argument after the object name. This example writes 2 holding registers starting at address 501 and reads 4 holding registers starting at address 11250. The `readCount` is `4`.

Example: `writeRead(m,501,[1024 512],11250,4)`

Data Types: `double`

`serverId` — Address of the server to send the command to
double

Address of the server to send the command to, specified as a double. Server ID must be specified after the object name, write address, values, read address, and read count. If you do not specify a `serverId`, the default of `1` is used. Valid values are `0–247`, with `0` being the broadcast address. This example writes 3 holding registers starting at address 400 and reads 4 holding registers starting at address 52008 from server ID 6.

Example: `writeRead(m,400,[1024 512 680],52008,4,6)`

Data Types: `double`

`writePrecision` — Data format of the holding register being written to on the MODBUS server
character vector | string

Data format of the holding register being written to on the MODBUS server, specified as a character vector or string. `writePrecision` must be specified after the write address and values. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

Note that `writePrecision` does not refer to the return type, which is always `'double'`. It specifies how to interpret the register data.

This example writes 3 holding registers starting at address 400 and reads 4 holding registers starting at address 52008 from server ID 6. It also specifies a `writePrecision` of `'uint64'`.

Example: `writeRead(m,400,[1024 512 680],'uint64',52008,4,'uint32',6)`

Data Types: `char`

`readPrecision` — Data format of the holding register being read from on the MODBUS server

character vector | string

Data format of the holding register being read from on the MODBUS server, specified as a character vector or string. `readPrecision` must be specified after the read address, and read count. Valid values are `'uint16'`, `'int16'`, `'uint32'`, `'int32'`, `'uint64'`, `'int64'`, `'single'`, and `'double'`. This argument is optional, and the default is `'uint16'`.

Note that `readPrecision` does not refer to the return type, which is always `'double'`. It specifies how to interpret the register data.

This example writes 3 holding registers starting at address 400 and reads 4 holding registers starting at address 52008 from server ID 6. It also specifies a `readPrecision` of `'uint32'`.

Example: `writeRead(m,400,[1024 512 680],'uint64',52008,4,'uint32',6)`

Data Types: `char`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`modbus` | `read` | `write` | `maskWrite`

Topics

“Create a MODBUS Connection” on page 11-3
“Configure Properties for MODBUS Communication” on page 11-5
“Write and Read Multiple Holding Registers” on page 11-16

Introduced in R2017a

Properties

ActualLocation

Configuration store file used by IVI configuration store object

Description

`ActualLocation` reflects the location of the IVI configuration store actually being used. It is either the master configuration store, or the `ProcessLocation` if an alternative to the master store was specified when the IVI configuration store object was created.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Character vector

Values

The default value is the master configuration store.

See Also

Functions

`commit`

Properties

MasterLocation, ProcessLocation

Alias

Alias of resource name for VISA instrument

Description

`Alias` indicates the alias for the resource name for a VISA instrument. When you create a VISA object, you can specify either the resource name for a VISA instrument or an alias for the resource name. If an alias is specified, `Alias` is automatically assigned the value specified in the VISA function. If a resource name is specified and the resource name has an alias, `Alias` is updated with the alias value. If the resource name does not have an alias, `Alias` is an empty character vector.

Characteristics

Usage	VISA object
Read only	Always
Data type	Character vector

Values

The default value is an empty character vector.

Remarks

You set the alias for a resource name using vender-supplied tools. You do not set an alias in the MATLAB workspace. When you create the VISA object in the MATLAB workspace, the `Alias` property of the object takes on the value of the resource name alias. You do not directly set the value of this property.

National Instruments' Measurement & Automation Explorer (MAX) is one example of a graphical interface tool for setting a VISA alias in NI-VISA. In this tool, select **Tools** > **NI-VISA** > **Alias Editor** to edit, add, or clear aliases. When you have your aliases defined, you can use them in the MATLAB workspace to access your resources.

See Also

Functions

`visa`

Properties

RsrcName

BaudRate

Specify bit transmit rate

Description

You configure BaudRate as bits per second. The transferred bits include the start bit, the data bits, the parity bit (if used), and the stop bits. However, only the data bits are stored.

The baud rate is the rate at which information is transferred in a communication channel. In the serial port context, "9600 baud" means that the serial port is capable of transferring a maximum of 9600 bits per second. If the information unit is one baud (one bit), then the bit rate and the baud rate are identical. If one baud is given as 10 bits, (for example, eight data bits plus two framing bits), the bit rate is still 9600 but the baud rate is 9600/10, or 960. You always configure BaudRate as bits per second. Therefore, in the above example, set BaudRate to 9600.

Note Both the computer and the instrument must be configured to the same baud rate before you can successfully read or write data.

Your system computes the acceptable rates by taking the baud base, which is determined by your serial port, and dividing it by a positive whole number divisor . The system will try to find the best match by modifying the divisor. For example, if:

```
baud base = 115200 bits per second
divisors = 1,2,3,4,5….
Possible BaudRates = 115200, 57600, 38400, 28800, 23040….
```

Your system may further limit the available baud rates to conform to specific conventions or standards. In the above example, for instance, 23040 bits/sec may not be available on all systems.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	Double

Values

The default value is 9600 bits per second.

Examples

This example shows how to set the baud rate for a serial port object.

Create a serial port object associated with the COM1 port. The oscilloscope you are connecting to over the serial port is configured to a baud rate of 115200 and a carriage return terminator, so set the serial port object to those values.

```
s = serial('COM1');
s.Baudrate = 115200;
s.Terminator = 'CR';
```

See Also

Properties

DataBits, Parity, StopBits

BoardIndex

Specify index number of interface board

Description

You configure BoardIndex to be the index number of the GPIB board, USB board, or TCP/IP board associated with your instrument. When you create a GPIB, VISA-GPIB, VISA-GPIB-VXI, VISA-TCPIP, or VISA-USB object, BoardIndex is automatically assigned the value specified in the gpib or visa function.

For GPIB objects, the Name property is automatically updated to reflect the BoardIndex value. For VISA-GPIB, VISA-GPIB-VXI, VISA-TCPIP, or VISA-USB objects, the Name and RsrcName properties are automatically updated to reflect the BoardIndex value.

You can configure BoardIndex only when the object is disconnected from the instrument. You disconnect a connected object with the fclose function. A disconnected object has a Status property value of closed.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-GPIB-VXI, VISA-TCPIP, VISA-USB
Read only	While open (GPIB, VISA-GPIB), always (VISA-GPIB-VXI, VISA-TCPIP, VISA-USB)
Data type	Double

Values

The value is defined after the instrument object is created.

Examples

Suppose you create a VISA-GPIB object associated with board 4, primary address 1, and secondary address 8.

```
vg = visa('keysight','GPIB4::1::8::INSTR');
```

The BoardIndex, Name, and RsrcName properties reflect the GPIB board index number.

```
vg.BoardIndex
ans =
    [4]

vg.Name
ans =
    'VISA-GPIB4-1-8'

vg,.RsrcName
ans =
        'GPIB4::1::8::INSTR'
```

See Also

Functions

`fclose`, `gpib`, `visa`

Properties

Name, RsrcName, Status

BreakInterruptFcn

Specify callback function to execute when break-interrupt event occurs

Description

You configure `BreakInterruptFcn` to execute a callback function when a break-interrupt event occurs. A break-interrupt event is generated by the serial port when the received data is in an off (space) state longer than the transmission time for one byte.

Note A break-interrupt event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a break-interrupt event occurs, the record file records this information:

- The event type as `BreakInterrupt`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Characteristics

Usage	Serial port
Read only	Never
Data type	Callback function

Values

The default value is an empty character vector.

See Also

Functions

`record`

Properties

RecordStatus

BusManagementStatus

State of GPIB bus management lines

Description

`BusManagementStatus` is a structure array that contains the fields `Attention`, `InterfaceClear`, `RemoteEnable`, `ServiceRequest`, and `EndOrIdentify`. These fields indicate the state of the Attention (ATN), Interface Clear (IFC), Remote Enable (REN), Service Request (SRQ) and End Or Identify (EOI) GPIB lines.

`BusManagementStatus` can be `on` or `off` for any of these fields. If `BusManagementStatus` is `on`, the associated line is asserted. If `BusManagementStatus` is `off`, the associated line is unasserted.

Characteristics

Usage	GPIB
Read only	Always
Data type	Structure

Values

`off`	The GPIB line is unasserted
`on`	The GPIB line is asserted

The default value is instrument dependent.

Examples

Create the GPIB object `g` associated with a National Instruments board, and connect `g` to a Tektronix TDS 210 oscilloscope.

```
g = gpib('ni',0,0);
fopen(g)
```

Write the `*STB?` command, which queries the instrument's status byte register, and then return the state of the bus management lines with the `BusManagementStatus` property.

```
fprintf(g,'*STB?')
g.BusManagementStatus
ans =
         Attention: 'off'
    InterfaceClear: 'off'
      RemoteEnable: 'on'
    ServiceRequest: 'off'
     EndOrIdentify: 'on'
```

REN is asserted because the system controller placed the scope in the remote enable mode, while EOI is asserted to mark the end of the command.

Now read the result of the *STB? command, and return the state of the bus management lines.

```
out = fscanf(g)
out =
0
g.busmanagementstatus
ans =
         Attention: 'off'
    InterfaceClear: 'off'
      RemoteEnable: 'on'
    ServiceRequest: 'off'
     EndOrIdentify: 'off'

fclose(g)
delete(g)
clear g
```

ByteOrder

Specify byte order of instrument

Description

You configure `ByteOrder` to be `littleEndian` or `bigEndian`. If `ByteOrder` is `littleEndian`, then the instrument stores the first byte in the first memory address. If `ByteOrder` is `bigEndian`, then the instrument stores the last byte in the first memory address.

For example, suppose the hexadecimal value 4F52 is to be stored in instrument memory. Because this value consists of two bytes, 4F and 52, two memory locations are used. Using big-endian format, 4F is stored first in the lower storage address. Using little-endian format, 52 is stored first in the lower storage address.

Note You should configure `ByteOrder` to the appropriate value for your instrument before performing a read or write operation. Refer to your instrument documentation for information about the order in which it stores bytes.

You can set this property on interface objects such as TCP/IP or GPIB. In this example, a TCP/IP object, `Tobj`, is set to `bigEndian` by default, and you change it to `littleEndian`.

```
Tobj.ByteOrder = 'littleEndian'
```

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Character vector

Values

`littleEndian`	The byte order of the instrument is little-endian. Default for `serial`, `gpib`, and `visa` objects.
`bigEndian`	The byte order of the instrument is big-endian. Default for `tcpip` and `udp` objects.

Examples

This example shows how to set the byte order for a TCP/IP object.

Create a TCP/IP object associated with the host 127.0.0.1 and port 4000. Change the byte order from the default of `bigEndian` to `littleEndian`.

```
t = tcpip('127.0.0.1', 4000);
t.ByteOrder = 'littleEndian';
```

See Also

Properties

Status

BytesAvailable

Number of bytes available in input buffer

Description

`BytesAvailable` indicates the number of bytes currently available to be read from the input buffer. The property value is continuously updated as the input buffer is filled, and is set to `0` after the `fopen` function is issued.

You can make use of `BytesAvailable` only when reading data asynchronously. This is because when reading data synchronously, control is returned to the MATLAB Command Window only after the input buffer is empty. Therefore, the `BytesAvailable` value is always 0.

The `BytesAvailable` value can range from zero to the size of the input buffer. Use the `InputBufferSize` property to specify the size of the input buffer. Use the `ValuesReceived` property to return the total number of values read.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Double

Values

The value can range from zero to the size of the input buffer. The default value is `0`.

See Also

Functions

`fopen`

Properties

InputBufferSize, TransferStatus, ValuesReceived

BytesAvailableFcn

Specify callback function to execute when specified number of bytes are available in input buffer, or terminator is read

Description

You configure `BytesAvailableFcn` to execute a callback function when a bytes-available event occurs. A bytes-available event occurs when the number of bytes specified by the `BytesAvailableFcnCount` property is available in the input buffer, or after a terminator is read, as determined by the `BytesAvailableFcnMode` property.

Note A bytes-available event can be generated only for asynchronous read operations.

If the `RecordStatus` property value is `on`, and a bytes-available event occurs, the record file records this information:

- The event type as `BytesAvailable`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Note You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Callback function

Values

The default value is an empty character vector.

Examples

Create the serial port object `s` on a Windows machine for a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1.

```
s = serial('COM1');
```

Configure `s` to execute the callback function `instrcallback` when 40 bytes are available in the input buffer.

```
s.BytesAvailableFcnCount = 40;
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcn = @instrcallback;
```

Connect s to the oscilloscope.

```
fopen(s)
```

Write the *IDN? command, which instructs the scope to return identification information. Because the default value for the ReadAsyncMode property is continuous, data is read as soon as it is available from the instrument.

```
fprintf(s,'*IDN?')
```

The resulting output from instrcallback is shown below.

```
BytesAvailable event occurred at 18:33:35 for the object:
Serial-COM1.
```

56 bytes are read and instrcallback is called once. The resulting display is shown above.

```
s.BytesAvailable
ans =
    56
```

Suppose you remove 25 bytes from the input buffer and issue the MEASUREMENT? command, which instructs the scope to return its measurement settings.

```
out = fscanf(s,'%c',25);
fprintf(s,'MEASUREMENT?')
```

The resulting output from instrcallback is shown below.

```
BytesAvailable event occurred at 18:33:48 for the object:
Serial-COM1.

BytesAvailable event occurred at 18:33:48 for the object:
Serial-COM1.
```

There are now 102 bytes in the input buffer, 31 of which are left over from the *IDN? command. instrcallback is called twice; once when 40 bytes are available and once when 80 bytes are available.

```
s.BytesAvailable
ans =
    102
```

See Also

Functions

record

Properties

BytesAvailableFcnCount, BytesAvailableFcnMode, EOSCharCode, RecordStatus, Terminator, TransferStatus

BytesAvailableFcnCount

Specify number of bytes that must be available in input buffer to generate bytes-available event

Description

You configure `BytesAvailableFcnCount` to the number of bytes that must be available in the input buffer before a bytes-available event is generated.

Use the `BytesAvailableFcnMode` property to specify whether the bytes-available event occurs after a certain number of bytes are available or after a terminator is read.

The bytes-available event executes the callback function specified for the `BytesAvailableFcn` property.

You can configure `BytesAvailableFcnCount` only when the object is disconnected from the instrument. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	Any instrument object
Read only	While open
Data type	Double

Values

The default value is `48`.

See Also

Functions

`fclose`

Properties

BytesAvailableFcn, BytesAvailableFcnMode, EOSCharCode, Status, Terminator

BytesAvailableFcnMode

Specify whether bytes-available event is generated after specified number of bytes are available in input buffer, or after terminator is read

Description

For serial port, TCPIP, UDP, or VISA-serial objects, you can configure `BytesAvailableFcnMode` to be `terminator` or `byte`. For all other instrument objects, you can configure `BytesAvailableFcnMode` to be `eosCharCode` or `byte`.

If `BytesAvailableFcnMode` is `terminator`, a bytes-available event occurs when the terminator specified by the `Terminator` property is read. If `BytesAvailableFcnMode` is `eosCharCode`, a bytes-available event occurs when the End-Of-String character specified by the `EOSCharCode` property is read. If `BytesAvailableFcnMode` is `byte`, a bytes-available event occurs when the number of bytes specified by the `BytesAvailableFcnCount` property is available.

The bytes-available event executes the callback function specified for the `BytesAvailableFcn` property.

You can configure `BytesAvailableFcnMode` only when the object is disconnected from the instrument. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	Any instrument object
Read only	While open
Data type	Character vector

Values

Serial, TCPIP, UDP, and VISA-Serial

Default value is enclosed in braces ({}).

`{terminator}`	A bytes-available event is generated when the terminator is reached.
`byte`	A bytes-available event is generated when the specified number of bytes available.

GPIB, VISA-GPIB, VISA-VXI, and VISA-GPIB-VXI

Default value is enclosed in braces ({}).

`{eosCharCode}`	A bytes-available event is generated when the EOS (End-Of-String) character is reached.
`byte`	A bytes-available event is generated when the specified number of bytes is available.

See Also

Functions

`fclose`

Properties

BytesAvailableFcn, BytesAvailableFcnCount, EOSCharCode, Status, Terminator

BytesToOutput

Number of bytes currently in output buffer

Description

`BytesToOutput` indicates the number of bytes currently in the output buffer waiting to be written to the instrument. The property value is continuously updated as the output buffer is filled and emptied, and is set to `0` after the `fopen` function is issued.

You can make use of `BytesToOutput` only when writing data asynchronously. This is because when writing data synchronously, control is returned to the MATLAB Command Window only after the output buffer is empty. Therefore, the `BytesToOutput` value is always `0`.

Use the `ValuesSent` property to return the total number of values written to the instrument.

Note If you attempt to write out more data than can fit in the output buffer, then an error is returned and `BytesToOutput` is `0`. You specify the size of the output buffer with the `OutputBufferSize` property.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Double

Values

The default value is `0`.

See Also

Functions

`fopen`

Properties

OutputBufferSize, TransferStatus, ValuesSent

ChassisIndex

Specify index number of VXI chassis

Description

You configure `ChassisIndex` to be the index number of the VXI chassis associated with your instrument.

When you create a VISA-VXI or VISA-GPIB-VXI object, `ChassisIndex` is automatically assigned the value specified in the `visa` function. For both object types, the `Name` and `RsrcName` properties are automatically updated to reflect the `ChassisIndex` value.

You can configure `ChassisIndex` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	While open
Data type	double

Values

The value is defined after the instrument object is created.

Examples

Suppose you create a VISA-GPIB-VXI object associated with chassis 0 and logical address 32.

```
v = visa('keysight','GPIB-VXI0::32::INSTR');
```

The `ChassisIndex`, `Name`, and `RsrcName` properties reflect the VXI chassis index number.

```
v.ChassisIndex
ans =
    [0]

v.Name
ans =
    'VISA-GPIB-VXI0-32'

v.RsrcName
ans =
    'GPIB-VXI0::32::INSTR'
```

See Also

Functions

`fclose`, `visa`

Properties

Name, RsrcName, Status

CompareBits

Specify number of bits that must match EOS character to complete read operation, or to assert EOI line

Description

You can configure `CompareBits` to be `7` or `8`. If `CompareBits` is `7`, the read operation completes when a byte that matches the low seven bits of the End-Of-String (EOS) character is received. The End Or Identify (EOI) line is asserted when a byte that matches the low seven bits of the EOS character is written. If `CompareBits` is `8`, the read operation completes when a byte that matches all eight bits of the EOS character is received. The EOI line is asserted when a byte that matches all eight bits of the EOS character is written.

You can specify the EOS character with the `EOSCharCode` property. You can specify when the EOS character is used (read operation, write operation, or both) with the `EOSMode` property.

Characteristics

Usage	GPIB
Read only	Never
Data type	Double

Values

Default value is enclosed in braces ({}).

{8}	Compare all eight EOS bits.
7	Compare the lower seven EOS bits.

See Also

Properties

EOSCharCode, EOSMode

ConfirmationFcn

Specify callback function to execute when confirmation event occurs

Description

You configure `ConfirmationFcn` to execute a callback function when a confirmation event occurs.

A confirmation event is generated when the command written to the instrument results in the instrument being configured to a different value than it was sent.

Note A confirmation event can be generated only when the object is connected to the instrument with `connect`.

Characteristics

Usage	Device
Read only	Never
Data type	Callback

Values

The default value is an empty character vector.

See Also

Functions

`connect`

DataBits

Specify number of data bits to transmit

Description

You can configure `DataBits` to be `5`, `6`, `7`, or `8`. Data is transmitted as a series of five, six, seven, or eight bits with the least significant bit sent first. At least seven data bits are required to transmit ASCII characters. Eight bits are required to transmit binary data. Five and six bit data formats are used for specialized communication equipment.

Note Both the computer and the instrument must be configured to transmit the same number of data bits.

In addition to the data bits, the serial data format consists of a start bit, one or two stop bits, and possibly a parity bit. You specify the number of stop bits with the `StopBits` property, and the type of parity checking with the `Parity` property.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	Double

Values

`DataBits` can be `5`, `6`, `7`, or `8`. The default value is `8`.

See Also

Properties

Parity, StopBits

DatagramAddress

IP dotted decimal address of received datagram sender

Description

`DatagramAddress` is the datagram sender IP address of the next datagram to be read from the input buffer. An example of an IP dotted decimal address character vector is 144.212.100.10.

When you read a datagram from the input buffer, `DatagramAddress` is updated.

Characteristics

Usage	UDP
Read only	Always
Data type	Character vector

Values

The default value is `''`.

See Also

Functions

`udp`

Properties

DatagramPort, RemotePort

DatagramPort

Port number of datagram sender

Description

`DatagramPort` is the port number of the next datagram to be read from the input buffer. When you read a datagram from the input buffer, `DatagramPort` is updated.

Characteristics

Usage	UDP
Read only	Never
Data type	Double

Values

The default value is `[]`.

See Also

Functions

`udp`

Properties

DatagramAddress

DatagramReceivedFcn

Specify callback function to execute when datagram is received

Description

You configure `DatagramReceivedFcn` to execute a callback function when a datagram has been received. The callback executes when a complete datagram is received in the input buffer.

Note A datagram-received event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a datagram-received event occurs, the record file records this information:

- The event type as `DatagramReceived`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Characteristics

Usage	UDP
Read only	Never
Data type	Callback

Values

The default value is `''`.

See Also

Functions

`readasync`, `udp`

Properties

DatagramAddress, DatagramPort, ReadAsyncMode

DatagramTerminateMode

Configure terminate read mode when reading datagrams

Description

`DatagramTerminateMode` defines how `fread` and `fscanf` read operations terminate. You can configure `DatagramTerminateMode` to be `on` or `off`.

If `DatagramTerminateMode` is `on`, the read operation terminates when a datagram is read. When `DatagramTerminateMode` is `off`, `fread` and `fscanf` read across datagram boundaries.

Characteristics

Usage	UDP
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{on}`	The read operation terminates when a datagram is read.
`off`	The read operation spans datagram boundaries.

See Also

Functions

`fread`, `fscanf`, `udp`

DataTerminalReady

Specify state of DTR pin

Description

You can configure `DataTerminalReady` to be `on` or `off`. If `DataTerminalReady` is `on`, the Data Terminal Ready (DTR) pin is asserted. If `DataTerminalReady` is `off`, the DTR pin is unasserted.

In normal usage, the DTR and Data Set Ready (DSR) pins work together, and are used to signal if instruments are connected and powered. However, there is nothing in the RS-232 or the RS-485 standard that states the DTR pin must be used in any specific way. For example, DTR and DSR might be used for handshaking. You should refer to your instrument documentation to determine its specific pin behavior.

You can return the value of the DSR pin with the `PinStatus` property. Handshaking is described in “Control Pins” on page 6-5.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{on}`	The DTR pin is asserted.
`off`	The DTR pin is unasserted.

See Also

Properties

FlowControl, PinStatus

DriverName

Specify name of driver used to communicate with instrument

Description

For device objects with a `DriverType` property value of `MATLAB Instrument Driver`, the `DriverName` property specifies the name of the MATLAB instrument driver that contains the supported instrument commands.

For device objects with a `DriverType` property value of `VXIplug&play` or `IVI-C`, the `DriverName` is the name of the VXI*plug&play* or IVI-C driver, respectively.

Characteristics

Usage	Device
Read only	Always
Data type	Character vector

Values

`DriverName` is defined at device object creation.

See Also

Properties

DriverType

DriverSessions

Array of driver sessions contained in IVI configuration store

Description

`DriverSessions` identifies all the driver sessions in the IVI configuration store. Each driver session maps a software module to a hardware asset and its `IOResourceDescriptor`. A driver session also determines default settings and behavior for its software module.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

See Also

Properties

HardwareAssets, SoftwareModules

DriverType

Specify type of driver used to communicate with instrument

Description

`DriverType` can be `MATLAB interface object`, `MATLAB VXIplug&play`, or `MATLAB IVI`. If `DriverType` is `MATLAB interface object`, an interface object is used to communicate with the instrument. If `DriverType` is `MATLAB VXIplug&play`, a VXI*plug&play* driver is used to communicate with the instrument. If `DriverType` is `MATLAB IVI`, an IVI driver is used to communicate with the instrument.

Characteristics

Usage	Device
Read only	Always
Data type	Character vector

Values

The `DriverType` value is defined at the device object creation. `DriverType` can be `MATLAB interface object`, `MATLAB VXIplug&play`, or `MATLAB IVI`.

See Also

Properties

DriverName

EOIMode

Specify if EOI line is asserted at end of write operation

Description

You can configure `EOIMode` to be `on` or `off`. If `EOIMode` is `on`, the End Or Identify (EOI) line is asserted at the end of a write operation. If `EOIMode` is `off`, the EOI line is not asserted at the end of a write operation. `EOIMode` applies to both binary and text write operations.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{on}`	The EOI line is asserted at the end of a write operation.
`off`	The EOI line is not asserted at the end of a write operation.

See Also

Properties

BusManagementStatus

EOSCharCode

Specify EOS character

Description

You can configure EOSCharCode to an integer value ranging from 0 to 255, or to the equivalent ASCII character. For example, to configure EOSCharCode to a carriage return, you specify the value to be CR or 13.

EOSCharCode replaces \n wherever it appears in the ASCII command sent to the instrument. Note that %s\n is the default format for the fprintf function.

For many practical applications, you will configure both EOSCharCode and the EOSMode property. EOSMode specifies when the EOS character is used. If EOSMode is write or read&write (writing is enabled), the EOI line is asserted every time the EOSCharCode value is written to the instrument. If EOSMode is read or read&write (reading is enabled), then the read operation might terminate when the EOSCharCode value is detected. For GPIB objects, the CompareBits property specifies the number of bits that must match the EOS character to complete a read or write operation.

To see how EOSCharCode and EOSMode work together, refer to the example on page 25-36 given in the EOSMode property description.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-USB
Read only	Never
Data type	ASCII value

Values

An integer value ranging from 0 to 255 or the equivalent ASCII character. The default value is LF, which corresponds to a line feed.

See Also

Functions

fprintf

Properties

CompareBits, EOSMode

EOSMode

Specify when EOS character is written or read

Description

For GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, and VISA-USB objects, you can configure `EOSMode` to be `none`, `read`, `write`, or `read&write`.

If `EOSMode` is `none`, the End-Of-String (EOS) character is ignored. If `EOSMode` is `read`, the EOS character is used to terminate a read operation. If `EOSMode` is `write`, the EOS character is appended to the ASCII command being written whenever `\n` is encountered. When the EOS character is written to the instrument, the End Or Identify (EOI) line is asserted. If `EOSMode` is `read&write`, the EOS character is used in both read and write operations.

The EOS character is specified by the `EOSCharCode` property. For GPIB objects, the `CompareBits` property specifies the number of bits that must match the EOS character to complete a read operation, or to assert the EOI line.

Rules for Completing a Read Operation

For any `EOSMode` value, the read operation completes when

- The EOI line is asserted.
- Specified number of values is read.
- A timeout occurs.

Additionally, if `EOSMode` is `read` or `read&write` (reading is enabled), then the read operation can complete when the `EOSCharCode` property value is detected.

Rules for Completing a Write Operation

Regardless of the `EOSMode` value, a write operation completes when

- The specified number of values is written.
- A timeout occurs.

Additionally, if `EOSMode` is `write` or `read&write`, the EOI line is asserted each time the `EOSCharCode` property value is written to the instrument.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-USB
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{none}`	The EOS character is ignored.
`read`	The EOS character is used for each read operation.
`write`	The EOS character is used for each write operation.
`read&write`	The EOS character is used for each read and write operation.

Examples

Suppose you input a nominal voltage signal of 2.0 volts into a function generator, and read back the voltage value using `fscanf`.

```
g = gpib('ni',0,1);
fopen(g)
fprintf(g,'Volt?')
out = fscanf(g)
out =
+2.00000E+00
```

The `EOSMode` and `EOSCharCode` properties are configured to terminate the read operation when an `E` character is encountered.

```
g.EOSMode = 'read'
g.EOSCharCode = 'E'
fprintf(g,'Volt?')
out = fscanf(g)
out =
+2.00000
```

See Also

Properties

CompareBits, EOIMode, EOSCharCode

ErrorFcn

Specify callback function to execute when error event occurs

Description

You configure `ErrorFcn` to execute a callback function when an error event occurs.

Note An error event is generated only for asynchronous read and write operations.

An error event is generated when a timeout occurs. A timeout occurs if a read or write operation does not successfully complete within the time specified by the `Timeout` property. An error event is not generated for configuration errors such as setting an invalid property value.

If the `RecordStatus` property value is `on`, and an error event occurs, the record file records this information:

- The event type as `Error`
- The error message
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Callback function

Values

The default value is an empty character vector.

See Also

Functions

`record`

Properties

RecordStatus, Timeout

FlowControl

Specify data flow control method to use

Description

You can configure FlowControl to be none, hardware, or software. If FlowControl is none, then data flow control (handshaking) is not used. If FlowControl is hardware, then hardware handshaking is used to control data flow. If FlowControl is software, then software handshaking is used to control data flow.

Hardware handshaking typically utilizes the Request to Send (RTS) and Clear to Send (CTS) pins to control data flow. Software handshaking uses control characters (Xon and Xoff) to control data flow. To learn more about hardware and software handshaking, refer to “Use Serial Port Control Pins” on page 6-22.

You can return the value of the CTS pin with the PinStatus property. You can specify the value of the RTS pin with the RequestToSend property. However, if FlowControl is hardware, and you specify a value for RequestToSend, then that value might not be honored.

If you set the FlowControl property to hardware on a serial object, and a hardware connection is not detected, the fwrite and the fprintf functions will return an error message. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Notes If you want to check to see if the device is asserting that it is ready to receive data, set the FlowControl to none. Once you connect to the device check the PinStatus structure for ClearToSend. If ClearToSend is off, there is a problem on the remote device side. If ClearToSend is on, there is a hardware FlowControl device prepared to receive data and you can execute fprintf and fwrite.

Although you might be able to configure your instrument for both hardware handshaking and software handshaking at the same time, the toolbox does not support this behavior.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces ({}).

{none}	No flow control is used.
hardware	Hardware flow control is used.

software	Software flow control is used.

See Also

Properties

PinStatus, RequestToSend

HandshakeStatus

State of GPIB handshake lines

Description

`HandshakeStatus` is a structure array that contains the fields `DataValid`, `NotDataAccepted`, and `NotReadyForData`. These fields indicate the state of the Data Valid (DAV), Not Data Accepted (NDAC) and Not Ready For Data (NRFD) GPIB lines, respectively.

`HandshakeStatus` can be `on` or `off` for any of these fields. A value of `on` indicates the associated line is asserted. A value of `off` indicates the associated line is unasserted.

Characteristics

Usage	GPIB
Read only	Never
Data type	Structure

Values

`on`	The associated handshake line is asserted
`off`	The associated handshake line is unasserted

The default value is instrument dependent.

HardwareAssets

Array of hardware assets contained in IVI configuration store

Description

`HardwareAssets` specifies all hardware assets in the IVI configuration store, each hardware asset referencing an `IOResourceDescriptor`.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

Values

The default value is empty.

See Also

Properties

DriverSessions, SoftwareModules

HwIndex

Hardware index of device group object

Description

Every device group object contained by a device object has an associated hardware index that is used to reference that device group object. For example, to configure property values for an individual device group object, you must reference the group object through its property name and the appropriate `HwIndex` value.

`HwIndex` provides a convenient way to programmatically access device group objects.

Characteristics

Usage	Device Group
Read only	Always
Data type	Double

Values

The default value is defined at the device group object creation.

See Also

Properties

HwName

HwName

Hardware name of device group object

Description

Every device group object contained by a device object has an associated hardware name that can be used to reference that device group object.

`HwName` provides a convenient way to programmatically access device group objects.

Characteristics

Usage	Device Group
Read only	Always
Data type	Character vector

Values

The default value is defined at the device group object creation.

See Also

Properties

HwIndex

InputBufferSize

Specify size of input buffer in bytes

Description

You configure `InputBufferSize` as the total number of bytes that can be stored in the software input buffer during a read operation.

A read operation is terminated if the amount of data stored in the input buffer equals the `InputBufferSize` value. You can read text data with the `fgetl`, `fgets`, or `fscanf` functions. You can read binary data with the `fread` function.

You can configure `InputBufferSize` only when the instrument object is disconnected from the instrument. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

If you configure `InputBufferSize` while there is data in the input buffer, then that data is flushed.

Characteristics

Usage	Any instrument object
Read only	While open
Data type	Double

Values

The default value is `512` bytes.

Examples

This example shows how to set the input buffer size for a serial port object. The `InputBufferSize` property specifies the total number of bytes that can be stored in the software input buffer during a read operation. By default, `InputBufferSize` is `512` bytes. There could be a case when you would want to increase it to higher than the default size.

Create a serial port object associated with the COM1 port. Set the input buffer size to 768 bytes.

```
s = serial('COM1');
s.InputBufferSize = 768;
```

See Also

Functions

`fclose`, `fgetl`, `fgets`, `fopen`, `fread`, `fscanf`

Properties

Status

InputDatagramPacketSize

Specify length of data received in a datagram

Description

Specify the length of the data received in a datagram. The size is the number of bytes of the packet's data buffer used to receive data.

If the data in a datagram packet is larger than the `InputDatagramPacketSize` the incoming data is truncated and some data is lost.

Characteristics

Usage	UDP
Read only	When open
Data type	Double

Values

You can specify a size, in bytes, between 1 and 65,535. The default value is 512.

See Also

Functions

`udp`

Properties

OutputDatagramPacketSize

InstrumentModel

Instrument model that object connects to

Description

`InstrumentModel` returns the information returned by the instrument identification command, e.g., `*IDN?`, `*ID?`. The instrument identification command is defined by the instrument driver.

Characteristics

Usage	Device
Read only	Always
Data type	Character vector

Values

`InstrumentModel` will be an empty character vector until the object is connected to the instrument with the `connect` function and the property value is queried.

See Also

Functions

`connect`, `get`

Interface

Interface object that communicates with instrument

Description

If `DriverType` is `MATLAB Instrument Driver`, then `Interface` is the interface object used to communicate with the instrument. If `DriverType` is `VXIplug&play` or `IVI-C`, then `Interface` is the handle to the VISA session that is used to communicate with the instrument.

Characteristics

Usage	Device
Read only	Always
Data type	Character vector

Values

`Interface` is defined at device object creation.

See Also

Properties

DriverType, LogicalName, RsrcName

InterfaceIndex

Specify USB interface number

Description

You configure `InterfaceIndex` to be the USB interface number.

The `Name` and `RsrcName` properties are automatically updated to reflect the `InterfaceIndex` value.

You can configure `InterfaceIndex` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-USB
Read only	While open
Data type	Double

See Also

Functions

`fclose`

Properties

Name, RsrcName

InterruptFcn

Specify callback function to execute when interrupt event occurs

Description

You configure `InterruptFcn` to execute a callback function when an interrupt event occurs. An interrupt event is generated when a VXI bus signal or a VXI bus interrupt is received from the instrument.

Note An interrupt event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and an interrupt event occurs, the record file records

- The event type as `Interrupt`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Characteristics

Usage	VISA-VXI
Read only	Never
Data type	Character vector

Values

The default value is an empty character vector.

See Also

Functions

`record`

Properties

RecordStatus

LANName

Specify LAN device name

Description

You configure `LANName` to be the LAN (Local Area Network) device name.

The `Name` and `RsrcName` properties are automatically updated to reflect the `LANName` value.

You can configure `LANName` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-TCPIP
Read only	While open
Data type	Character vector

See Also

Functions

`fclose`

Properties

Name, RsrcName

LocalHost

Specify local host

Description

LocalHost specifies the local host name or the IP dotted decimal address. An example dotted decimal address is 144.212.100.10. If you have only one address or you do not specify this property, the object uses the default IP address when you connect to the hardware with the fopen function.

You can configure LocalHost only when the object is disconnected from the hardware. You disconnect a connected object with the fclose function. A disconnected object has a Status property value of closed.

Characteristics

Usage	TCPIP, UDP
Read only	While open
Data type	Character vector

Values

The default value is an empty character vector.

See Also

Functions

fclose, fopen, tcpip, udp

Properties

LocalPort, RemoteHost, Status

LocalPort

Specify local host port for connection

Description

You configure LocalPort to be the port value of the local host. The default value is [].

If LocalPortMode is set to auto or if LocalPort is [], the property is assigned any free port when you connect the object to the hardware with the fopen function. If LocalPortMode is set to manual, the specified LocalPort value is used when you issue fopen. If you explicitly configure LocalPort, LocalPortMode is automatically set to manual.

You can configure LocalPort only when the object is disconnected from the hardware. You disconnect a connected object with the fclose function. A disconnected object has a Status property value of closed.

Characteristics

Usage	TCPIP, UDP
Read only	While open
Data type	Double

Values

The default value is [].

See Also

Functions

fclose, fopen, tcpip, udp

Properties

LocalHost, LocalPortMode, Status

LocalPortMode

Specify local host port selection mode

Description

`LocalPortMode` specifies the selection mode for the `LocalPort` property when you connect a TCPIP or UDP object.

If `LocalPortMode` is set to `auto`, the MATLAB workspace uses any free local port. If `LocalPortMode` is set to `manual`, the specified `LocalPort` value is used when you issue the `fopen` function. If you explicitly specify a value for `LocalPort`, `LocalPortMode` is automatically set to `manual`.

Characteristics

Usage	TCPIP, UDP
Read only	While open
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{auto}`	Use any free local port.
`manual`	Use the specified local port value.

See Also

Functions

`fclose`, `fopen`, `tcpip`, `udp`

Properties

LocalHost, LocalPort, Status

LogicalAddress

Specify logical address of VXI instrument

Description

For VISA-VXI and VISA-GPIB-VXI objects, you configure `LogicalAddress` to be the logical address of the VXI instrument. You must include the logical address as part of the resource name during object creation using the `visa` function.

The `Name` and `RsrcName` properties are automatically updated to reflect the `LogicalAddress` value.

You can configure `LogicalAddress` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	While open
Data type	Double

Values

The value is defined when the instrument object is created.

Examples

This example creates a VISA-VXI object associated with chassis 4 and logical address 1, and then returns the logical address.

```
vv = visa('keysight','VXI4::1::INSTR');
vv.LogicalAddress
ans =
     1
```

See Also

Functions

`fclose`, `visa`

Properties

Name, RsrcName, Status

LogicalName

Specify description of interface used to communicate with instrument

Description

For device objects with a `DriverType` property value of `MATLAB Instrument Driver`, the `LogicalName` property specifies the type of interface used to communicate with the instrument. For example, a `LogicalName` of `GPIB0-2` indicates that communication is through a GPIB board at index 0 with an instrument at primary address 2.

For device objects with a `DriverType` property value of `VXIplug&play`, the `LogicalName` is the resource name used to communicate with the instrument.

For device objects with a `DriverType` property value of `IVI-C`, the `LogicalName` is the `LogicalName` associated with the IVI-C driver.

Characteristics

Usage	Device
Read only	Always
Data type	Character vector

Values

`LogicalName` is defined at device object creation.

See Also

Properties

DriverType, Interface, RsrcName

LogicalNames

Array of logical names contained in IVI configuration store

Description

Each entry in `LogicalNames` identifies a logical name in the IVI configuration store. Each logical name references a driver session in the configuration store, and is used in creating device objects with the `icdevice` function.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

See Also

Functions

`icdevice`

Properties

DriverSessions

ManufacturerID

Specify manufacturer ID of USB instrument

Description

You configure `ManufacturerID` to be the manufacturer ID of the USB instrument.

The `Name` and `RsrcName` properties are automatically updated to reflect the `ManufacturerID` value.

You can configure `ManufacturerID` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-USB
Read only	While open
Data type	Character vector

See Also

Functions

`fclose`

Properties

Name, RsrcName

MappedMemoryBase

Base memory address of mapped memory

Description

MappedMemoryBase is the base address of the mapped memory used for low level read and write operations.

The memory address is returned as a character vector representing a hexadecimal value. For example, if the mapped memory base is 200000, then MappedMemoryBase returns 200000H. If no memory is mapped, MappedMemoryBase is 0H.

Use the memmap function to map the specified amount of memory in the specified address space (A16, A24, or A32) with the specified offset. Use the memunmap function to unmap the memory space.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	Character vector

Values

The default value is 0H.

Examples

Create the VISA-VXI object vv associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

Map 16 bytes in the A16 address space with no offset, and then return the base address of the mapped memory.

```
memmap(vv,'A16',0,16)
vv.MappedMemoryBase
ans =
     16737610H
```

See Also

Functions

memmap, memunmap

Properties

MappedMemorySize

MappedMemorySize

Size of mapped memory for low-level read and write operations

Description

`MappedMemorySize` indicates the amount of memory mapped for low-level read and write operations.

Use the `memmap` function to map the specified amount of memory in the specified address space (A16, A24, or A32) with the specified offset. Use the `memunmap` function to unmap the memory space.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	Double

Values

The default value is `0`.

Examples

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

Map 16 bytes in the A16 address space with no offset, and then return the size of the mapped memory.

```
memmap(vv,'A16',0,16)
vv.MappedMemorySize
ans =
     16
```

See Also

Functions

`memmap`, `memunmap`

Properties

MappedMemoryBase

MasterLocation

Full pathname of master configuration store file

Description

`MasterLocation` identifies the master (default) configuration store location.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Character vector

Values

The default value is set at IVI installation.

See Also

Properties

ActualLocation, ProcessLocation

MemoryBase

Base address of A24 or A32 space

Description

MemoryBase indicates the base address of the A24 or A32 space. The value is returned as a character vector representing a hexadecimal value.

All VXI instruments have an A16 address space that is 16 bits wide. There are also 24- and 32-bit wide address spaces known as A24 and A32. Some instruments require the additional memory associated with the A24 or A32 address space when the 64 bytes of A16 space are insufficient for performing necessary functions. A bit in the A16 address space is set allowing the instrument to recognize commands to its A24 or A32 space.

An instrument cannot use both the A24 and A32 address space. The address space is given by the MemorySpace property. If MemorySpace is A16, then MemoryBase is 0H.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	Character vector

Values

The default value is 0H.

Examples

Create the VISA-VXI object vv associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

The MemorySpace property indicates that the A24 memory space is supported.

```
vv.MemorySpace
ans =
A16/A24
```

The base address of the A24 space is

```
vv.MemoryBase
ans =
'200000H'
```

See Also

Properties

MemorySpace

MemoryIncrement

Specify whether VXI register offset increments after data is transferred

Description

You can configure MemoryIncrement to be block or FIFO. If MemoryIncrement is block, the memread and memwrite functions increment the offset after every read and write operation, and data is transferred from or to consecutive memory elements. If MemoryIncrement is FIFO, the memread and memwrite functions do not increment the VXI register offset, and data is always read from or written to the same memory element.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces ({}).

{block}	Increment the VXI register offset.
FIFO	Do not increment the VXI register offset.

Examples

Create the VISA-VXI object v associated with a VXI chassis with index 0, and an instrument with logical address 8.

```
v = visa('ni','VXI0::8::INSTR');
fopen(v)
```

Configure the hardware for a FIFO read and write operation.

```
v.MemoryIncrement = 'FIFO'
```

Write two values to the VXI register starting at offset 16. Because MemoryIncrement is FIFO, the VXI register offset does not change and both values are written to offset 16.

```
memwrite(v,[1984 2000],16,'uint32','A16')
```

Read the value at offset 16. The value returned is the second value written with the memwrite function.

```
memread(v,16,'uint32')
ans =
2000
```

Read two values starting at offset 16. Note that both values are read at offset 16.

```
memread(v,16,'uint32','A16',2);
ans =
2000
2000
```

Configure the hardware for a block read and write operation.

```
v.MemoryIncrement = 'block'
```

Write two values to the VXI register starting at offset 16. The first value is written to offset 16 and the second value is written to offset 20 because a `uint32` value consists of four bytes.

```
memwrite(v,[1984 2000],16,'uint32','A16')
```

Read the value at offset 16. The value returned is the first value written with the `memwrite` function.

```
memread(v,16,'uint32')
ans =
1984
```

Read two values starting at offset 16. The first value is read at offset 16 and the second value is read at offset 20.

```
memread(v,16,'uint32','A16',2);
ans =
1984
2000
```

See Also

Functions

`mempeek`, `mempoke`, `memread`, `memwrite`

MemorySize

Size of memory requested in A24 or A32 address space

Description

`MemorySize` indicates the size of the memory requested by the instrument in the A24 or A32 address space.

Some instruments use the A24 or A32 address space when the 64 bytes of A16 space are not enough for performing necessary functions. An instrument cannot use both the A24 and A32 address space. The address space is given by the `MemorySpace` property. If `MemorySpace` is `A16`, then `MemorySize` is `0`.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	Double

Values

The default value is `0`.

Examples

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

The `MemorySpace` property indicates that the A24 memory space is supported.

```
vv.MemorySpace
ans =
A16/A24
```

The size of the A24 space is

```
vv.MemorySize
ans =
262144
```

See Also

Properties

MemorySpace

MemorySpace

Address space used by instrument

Description

MemorySpace indicates the address space requested by the instrument. MemorySpace can be A16, A16/A24, or A16/A32. If MemorySpace is A16, the instrument uses only the A16 address space. If MemorySpace is A16/A24, the instrument uses the A16 and A24 address space. If MemorySpace is A16/A32, the instrument uses the A16 and A32 address space.

All VXI instruments have an A16 address space that is 16 bits wide. There are also 24- and 32-bit wide address spaces known as A24 and A32, respectively. Some instruments use this memory when the 64 bytes of A16 space are not enough for performing necessary functions. An instrument cannot use both the A24 and A32 address space.

The size of the memory is given by the MemorySize property. The base address of the memory is given by the MemoryBase property.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	Character vector

Values

Default value is enclosed in braces ({}).

{A16}	The instrument uses the A16 address space.
A16/A24	The instrument uses the A16 and A24 address space.
A16/A32	The instrument uses the A16 and A32 address space.

Examples

Create the VISA-VXI object vv associated with a VXI chassis with index 0, and a Keysight E1432A digitizer with logical address 130.

```
vv = visa('keysight','VXI0::130::INSTR');
fopen(vv)
```

Return the memory space supported by the instrument.

```
vv.MemorySpace
ans =
A16/A24
```

This value indicates that the instrument supports A24 memory space in addition to the A16 memory space.

See Also

Properties

MemoryBase, MemorySize

ModelCode

Specify model code of USB instrument

Description

You configure `ModelCode` to be the model code of the USB instrument.

The `Name` and `RsrcName` properties are automatically updated to reflect the `ModelCode` value.

You can configure `ModelCode` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-USB
Read only	While open
Data type	Character vector

See Also

Functions

`fclose`

Properties

Name, RsrcName

Name

Specify descriptive name for instrument object

Description

You configure Name to be a descriptive name for an instrument object.

When you create an instrument object, a descriptive name is automatically generated and stored in Name. However, you can change this value at any time. As shown below, the components of Name reflect the instrument object type and the input arguments you supply to the creation function.

Instrument Object	Default Value of Name
GPIB	GPIB and BoardIndex-PrimaryAddress-SecondaryAddress
serial port	Serial and Port
TCPIP	TCPIP and RemoteHost
UDP	UDP and RemoteHost
VISA-serial	VISA-Serial and Port
VISA-GPIB	VISA-GPIB and BoardIndex-PrimaryAddress-SecondaryAddress
VISA-VXI	VISA-VXI and ChassisIndex-LogicalAddress
VISA-GPIB-VXI	VISA-GPIB-VXI and ChassisIndex-LogicalAddress
VISA-TCPIP	VISA-TCPIP and BoardIndex-RemoteHost-LANName
VISA-RSIB	VISA-RSIB and RemoteHost
VISA-USB	VISA-USB and BoardIndex-ManufacturerID- ModelCode-SerialNumber-InterfaceIndex

If the secondary address is not specified when a GPIB or VISA-GPIB object is created, then Name does not include this component.

If you change the value of any property that is a component of Name (for example, Port or PrimaryAddress), then Name is automatically updated to reflect those changes.

Characteristics

Usage	Any instrument object
Read-only	Never
Data type	Character vector

Values

Name is automatically defined at object creation time. The value of Name depends on the specific instrument object you create.

Name (iviconfigurationstore)

Name of IVI configuration server

Description

`Name` identifies the name of the IVI configuration store server. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Character vector

NetworkRole

Specify server socket connection

Description

The `NetworkRole` property in the `tcpip` interface enables support for Server Sockets. It uses two values, `client` and `server`, to establish a connection as the client or the server.

The server sockets feature supports binary and ASCII transfers and supports a single remote connection.

Characteristics

Usage	TCPIP
Read only	While open
Data type	Character vector

Values

The default value is `client`.

`client`	Establish a TCP/IP connection as a client (default)
`server`	Establish a TCP/IP connection as a server

See Also

Functions

`fclose`, `fopen`, `tcpip`

How To

“Communicate Using TCP/IP Server Sockets” on page 7-34

ObjectVisibility

Control access to instrument object

Description

The ObjectVisibility property provides a way for application developers to prevent end-user access to the instrument objects created by their application. When an object's ObjectVisibility property is set to off, instrfind and instrreset do not return or delete those objects.

Objects that are not visible are still valid. If you have access to the object (for example, from within the file that creates it), then you can set and get its properties and pass it to any function that operates on instrument objects.

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces ({}).

{on}	Object is visible to instrfind and instrreset
off	Object is not visible from the command line (except by instrfindall)

Examples

The following statement creates an instrument object with its ObjectVisibility property set to off:

```
g = gpib('mcc',0,2,'ObjectVisibility','off');
instrfind
ans =
     []
```

However, since the object is in the workspace (g), you can access it.

```
g.ObjectVisibility
ans =

    off
```

See Also

Functions

`instrfind`, `instrfindall`, `instrreset`

OutputBufferSize

Specify size of output buffer in bytes

Description

You configure `OutputBufferSize` as the total number of bytes that can be stored in the software output buffer during a write operation.

An error occurs if the output buffer cannot hold all the data to be written. You write text data with the `fprintf` function. You write binary data with the `fwrite` function.

You can configure `OutputBufferSize` only when the instrument object is disconnected from the instrument. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	Any instrument object
Read only	While open
Data type	Double

Values

The default value is `512` bytes.

Examples

This example shows how to set the output buffer size for a serial port object. The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the instrument at once. By default, `OutputBufferSize` is `512` bytes. There could be a case when you would want to limit it to less than the default size.

Create a serial port object associated with the COM1 port. Set the output buffer size to 256 bytes.

```
s = serial('COM1');
s.OutputBufferSize = 256;
```

See Also

Functions

`fprintf`, `fwrite`

Properties

Status

OutputDatagramPacketSize

Specify length of data sent in a datagram

Description

Specify the length of the data sent in a datagram. The size is the number of bytes of the packet's data buffer used to send data.

If the data in a datagram packet is larger than the packet size the target device receives, some data is lost.

Characteristics

Usage	UDP
Read only	When open
Data type	Double

Values

You can specify a size, in bytes, between 1 and 65,535. The default value is 512.

See Also

Functions

udp

Properties

InputDatagramPacketSize

OutputEmptyFcn

Specify callback function to execute when output buffer is empty

Description

You configure `OutputEmptyFcn` to execute a callback function when an output-empty event occurs. An output-empty event is generated when the last byte is sent from the output buffer to the instrument.

Note An output-empty event can be generated only for asynchronous write operations.

If the `RecordStatus` property value is `on`, and an output-empty event occurs, the record file records this information:

- The event type as `OutputEmpty`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Callback function

Values

The default value is an empty character vector.

See Also

Functions

`record`

Properties

RecordStatus

Parent

Parent (device object) of device group object

Description

The parent of a device group object is defined as the device object that contains the device group object.

You can create a copy of the device object containing a particular device group object by returning the value of `Parent`. This copy can be treated like any other device object. For example, you can configure property values, connect to the instrument, and so on.

Characteristics

Usage	Device group
Read only	Always
Data type	Device object

Values

`Parent` is defined at device object creation.

Parity

Specify type of parity checking

Description

You can configure Parity to be none, odd, even, mark, or space. If Parity is none, parity checking is not performed and the parity bit is not transmitted. If Parity is odd, the number of mark bits (1s) in the data is counted, and the parity bit is asserted or unasserted to obtain an odd number of mark bits. If Parity is even, the number of mark bits in the data is counted, and the parity bit is asserted or unasserted to obtain an even number of mark bits. If Parity is mark, the parity bit is asserted. If Parity is space, the parity bit is unasserted.

Parity checking can detect errors of one bit only. An error in two bits might cause the data to have a seemingly valid parity, when in fact it is incorrect. To learn more about parity checking, refer to “Parity Bit” on page 6-8.

In addition to the parity bit, the serial data format consists of a start bit, between five and eight data bits, and one or two stop bits. You specify the number of data bits with the DataBits property, and the number of stop bits with the StopBits property.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces ({}).

{none}	No parity checking
odd	Odd parity checking
even	Even parity checking
mark	Mark parity checking
space	Space parity checking

Examples

This example shows how to set the parity for a serial port object.

Create a serial port object associated with the COM1 port. The default setting for Parity is none, so if you want to use parity checking, change the value to the type you want to use, for example, odd.

```
s = serial('COM1');
s.Parity = 'odd';
```

See Also

Properties

DataBits, StopBits

PinStatus

State of CD, CTS, DSR, and RI pins

Description

PinStatus is a structure array that contains the fields CarrierDetect, ClearToSend, DataSetReady and RingIndicator. These fields indicate the state of the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) and Ring Indicator (RI) pins, respectively. Refer to "Control Pins" on page 6-5 to learn more about these pins.

PinStatus can be on or off for any of these fields. A value of on indicates the associated pin is asserted. A value of off indicates the associated pin is unasserted. For serial port objects, a pin status event occurs when any of these pins changes its state. A pin status event executes the file specified by PinStatusFcn.

In normal usage, the Data Terminal Ready (DTR) and DSR pins work together, while the Request To Send (RTS) and CTS pins work together. You can specify the state of the DTR pin with the DataTerminalReady property. You can specify the state of the RTS pin with the RequestToSend property.

Refer to "Connect Two Modems" on page 6-22 for an example that uses PinStatus.

Characteristics

Usage	Serial port, VISA-serial
Read only	Always
Data type	Structure

Values

off	The associated pin is asserted
on	The associated pin is asserted

The default value is instrument dependent.

See Also

Properties

DataTerminalReady, PinStatusFcn, RequestToSend

PinStatusFcn

Specify callback function to execute when CD, CTS, DSR, or RI pin changes state

Description

You configure `PinStatusFcn` to execute a callback function when a pin status event occurs. A pin status event occurs when the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) or Ring Indicator (RI) pin changes state. A serial port pin changes state when it is asserted or unasserted. Information about the state of these pins is recorded in the `PinStatus` property.

Note A pin status event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a pin status event occurs, the record file records this information:

- The event type as `PinStatus`
- The pin that changed its state, and pin state as either `on` or `off`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Characteristics

Usage	Serial port
Read only	Never
Data type	Callback function

Values

The default value is an empty character vector.

See Also

Functions

`record`

Properties

PinStatus, RecordStatus

Port

Specify platform-specific serial port name

Description

You configure Port to be the name of a serial port on your platform. Port specifies the physical port associated with the object and the instrument.

When you create a serial port or VISA-serial object, Port is automatically assigned the port name specified for the serial or visa function.

You can configure Port only when the object is disconnected from the instrument. You disconnect an object with the fclose function. A disconnected object has a Status property value of closed.

Characteristics

Usage	Serial port, VISA-serial
Read only	While open
Data type	Character vector

Values

The value is determined when the instrument object is created.

Examples

Suppose you create a serial port and VISA-serial object associated with serial port COM1.

```
s = serial('COM1')
vs = visa('ni','ASRL1::INSTR')
```

The Port property values are given below.

```
[s vs].Port
ans =
    'COM1'
    'ASRL1'
```

See Also

Functions

fclose, serial, visa

Properties

Name, RsrcName, Status

PrimaryAddress

Specify primary address of GPIB instrument

Description

For GPIB and VISA-GPIB objects, you configure `PrimaryAddress` to be the GPIB primary address associated with your instrument. The primary address can range from 0 to 30, and you must specify it during object creation using the `gpib` or `visa` function. For VISA-GPIB-VXI objects, `PrimaryAddress` is read-only, and the value is returned automatically by the VISA interface after the object is connected to the instrument with the `fopen` function.

For GPIB and VISA-GPIB objects, the `Name` property is automatically updated to reflect the `PrimaryAddress` value. For VISA-GPIB objects, the `RsrcName` property is automatically updated to reflect the `PrimaryAddress` value.

You can configure `PrimaryAddress` only when the GPIB or VISA-GPIB object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-GPIB-VXI
Read only	While open (GPIB, VISA-GPIB), always (VISA-GPIB-VXI)
Data type	Double

Values

`PrimaryAddress` can range from 0 to 30. The value is determined when the instrument object is created.

Examples

This example creates a VISA-GPIB object associated with board 0, primary address 1, and secondary address 8, and then returns the primary address.

```
vg = visa('keysight','GPIB0::1::8::INSTR');
vg.PrimaryAddress
ans =
     1
```

See Also

Functions

`fclose`, `gpib`, `visa`

Properties

Name, RsrcName, Status

ProcessLocation

Configuration store file for process to use if master configuration store is not used

Description

`ProcessLocation` identifies an IVI configuration store being used as an alternative to the master configuration store. The use of an alternative is particular to each `iviconfigurationstore` object, and is specified when the object is created.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Character vector

Values

The default value is an empty character vector.

See Also

Functions

`iviconfigurationstore`

Properties

ActualLocation, MasterLocation

PublishedAPIs

Array of published APIs in IVI configuration store

Description

`PublishedAPIs` identifies the published APIs in the IVI configuration store server. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

ReadAsyncMode

Specify whether asynchronous read operation is continuous or manual

Description

You can configure `ReadAsyncMode` to be `continuous` or `manual`. If `ReadAsyncMode` is `continuous`, the object continuously queries the instrument to determine if data is available to be read. If data is available, it is automatically read and stored in the input buffer. If issued, the `readasync` function is ignored.

If `ReadAsyncMode` is `manual`, the object will not query the instrument to determine if data is available to be read. Instead, you must manually issue the `readasync` function to perform an asynchronous read operation. Because `readasync` checks for the terminator, this function can be slow. To increase speed, you should configure `ReadAsyncMode` to `continuous`.

Note If the instrument is ready to transmit data, then it will do so regardless of the `ReadAsyncMode` value. Therefore, if `ReadAsyncMode` is `manual` and a read operation is not in progress, then data can be lost. To guarantee that all transmitted data is stored in the input buffer, you should configure `ReadAsyncMode` to `continuous`.

You can determine the amount of data available in the input buffer with the `BytesAvailable` property. For either `ReadAsyncMode` value, you can bring data into the MATLAB workspace with one of the synchronous read functions such as `fscanf`, `fgetl`, `fgets`, or `fread`.

Characteristics

Usage	Serial port, TCPIP, UDP, VISA-serial
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{continuous}`	Continuously query the instrument to determine if data is available to be read.
`manual`	Manually read data from the instrument using the `readasync` function.

See Also

Functions

`fgetl`, `fgets`, `fread`, `fscanf`, `readasync`

Properties

BytesAvailable, InputBufferSize

RecordDetail

Specify amount of information saved to record file

Description

You can configure `RecordDetail` to be `compact` or `verbose`. If `RecordDetail` is `compact`, the number of values written to the instrument, the number of values read from the instrument, the data type of the values, and event information are saved to the record file. If `RecordDetail` is `verbose`, the data transferred to and from the instrument is also saved to the record file.

The verbose record file structure is shown in “Recording Information to Disk” on page 17-7.

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{compact}`	The number of values written to the instrument, the number of values read from the instrument, the data type of the values, and event information are saved to the record file.
`verbose`	The data written to the instrument, and the data read from the instrument are also saved to the record file.

See Also

Functions

`record`

Properties

RecordMode, RecordName, RecordStatus

RecordMode

Specify whether data and event information are saved to one or to multiple record files

Description

You can configure RecordMode to be overwrite, append, or index. If RecordMode is overwrite, then the record file is overwritten each time recording is initiated. If RecordMode is append, then data is appended to the record file each time recording is initiated. If RecordMode is index, a different record file is created each time recording is initiated, each with an indexed filename.

You can configure RecordMode only when the object is not recording. You terminate recording with the record function. A object that is not recording has a RecordStatus property value of off.

You specify the record filename with the RecordName property. The indexed filename follows a prescribed set of rules. Refer to “Specifying a File Name” on page 17-6 for a description of these rules.

Characteristics

Usage	Any instrument object
Read only	While recording
Data type	Character vector

Values

Default value is enclosed in braces ({}).

{overwrite}	The record file is overwritten.
append	Data is appended to the record file.
index	Multiple record files are written, each with an indexed filename.

Examples

Suppose you create the serial port object s on a Windows machine associated with the serial port COM1.

```
s = serial('COM1');
fopen(s)
```

Specify the record filename with the RecordName property, configure RecordMode to index, and initiate recording.

```
s.RecordName = 'myrecord.txt';
s.RecordMode = 'index';
record(s)
```

The record filename is automatically updated with an indexed filename after recording is turned off.

```
record(s,'off')
s.RecordName
ans =
myrecord01.txt
```

Disconnect `s` from the instrument, and remove `s` from memory and from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

Functions

`record`

Properties

RecordDetail, RecordName, RecordStatus

RecordName

Specify name of record file

Description

You configure `RecordName` to be the name of the record file. You can specify any value for `RecordName` — including a directory path — provided the filename is supported by your operating system.

The MATLAB software supports any filename supported by your operating system. However, if you access the file through the MATLAB workspace, you might need to specify the filename using single quotes. For example, suppose you name the record file `my record.txt`. To type this file at the MATLAB Command Window, you must include the name in quotes.

```
type('my record.txt')
```

You can specify whether data and event information are saved to one disk file or to multiple disk files with the `RecordMode` property. If `RecordMode` is `index`, then the filename follows a prescribed set of rules. Refer to “Specifying a File Name” on page 17-6 for a description of these rules.

You can configure `RecordName` only when the object is not recording. You terminate recording with the `record` function. An object that is not recording has a `RecordStatus` property value of `off`.

Characteristics

Usage	Any instrument object
Read only	While recording
Data type	Character vector

Values

The default record file name is `record.txt`.

See Also

Functions

`record`

Properties

RecordDetail, RecordMode, RecordStatus

RecordStatus

Status of whether data and event information are saved to record file

Description

You can configure `RecordStatus` to be `off` or `on` with the `record` function. If `RecordStatus` is `off`, then data and event information are not saved to a record file. If `RecordStatus` is `on`, then data and event information are saved to the record file specified by `RecordName`.

Use the `record` function to initiate or complete recording. `RecordStatus` is automatically configured to reflect the recording state.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{off}`	Data and event information are not written to a record file
`on`	Data and event information are written to a record file

See Also

Functions

`record`

Properties

RecordDetail, RecordMode, RecordName

RemoteHost

Specify remote host

Description

`RemoteHost` specifies the remote host name or IP dotted decimal address. An example dotted decimal address is 144.212.100.10.

For TCPIP objects, you can configure `RemoteHost` only when the object is disconnected from the hardware. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

For UDP objects, you can configure `RemoteHost` at any time. If the object is open, a warning is issued if the remote address is invalid.

Characteristics

Usage	TCPIP, UDP
Read only	While open (TCPIP), never (UDP)
Data type	Character vector

Values

The value is defined when you create the TCPIP or UDP object.

See Also

Functions

`fclose`, `fopen`, `tcpip`, `udp`

Properties

LocalHost, RemotePort, Status

RemotePort

Specify remote host port for connection

Description

You can configure `RemotePort` to be any port number between `1` and `65535`. The default value is `80` for TCPIP objects and `9090` for UDP objects.

For TCPIP objects, you can configure `RemotePort` only when the object is disconnected from the hardware. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

For UDP objects, you can configure `RemotePort` at any time.

Characteristics

Usage	TCPIP, UDP
Read only	While open (TCPIP), never (UDP)
Data type	Double

Values

Any port number between `1` and `65535`. The default value is `80` for TCPIP objects and `9090` for UDP objects.

See Also

Functions

`fclose`, `fopen`, `tcpip`, `udp`

Properties

RemoteHost, LocalPort, Status

RequestToSend

Specify state of RTS pin

Description

You can configure `RequestToSend` to be `on` or `off`. If `RequestToSend` is `on`, the Request to Send (RTS) pin is asserted. If `RequestToSend` is `off`, the RTS pin is unasserted.

In normal usage, the RTS and Clear to Send (CTS) pins work together, and are used as standard handshaking pins for data transfer. In this case, RTS and CTS are automatically managed by the DTE and DCE. However, there is nothing in the RS-232, or the RS-484 standard that states the RTS pin must to be used in any specific way. Therefore, if you manually configure the `RequestToSend` value, it is probably for nonstandard operations.

If your instrument does not use hardware handshaking in the standard way, and you need to manually configure `RequestToSend`, then you should configure the `FlowControl` property to `none`. Otherwise, the `RequestToSend` value that you specify might not be honored. Refer to your instrument documentation to determine its specific pin behavior.

You can return the value of the CTS pin with the `PinStatus` property. Handshaking is described in “Control Pins” on page 6-5.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{on}`	The RTS pin is asserted.
`off`	The RTS pin is unasserted.

See Also

Properties

FlowControl, PinStatus

Revision

IVI configuration store version

Description

`Revision` identifies the version of the IVI configuration store. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Character vector

RsrcName

Resource name for VISA instrument

Description

`RsrcName` indicates the resource name for a VISA instrument. When you create a VISA object, `RsrcName` is automatically assigned the value specified in the `visa` function.

The resource name is a symbolic name for the instrument. The resource name you supply to `visa` depends on the interface and has the format shown below. The components in brackets are optional and have a default value of 0, except port_number, which has a default value of 1.

Interface	Resource Name
VXI	VXI[chassis]::VXI_logical_address::INSTR
GPIB-VXI	GPIB-VXI[chassis]::VXI_logical_address::INSTR
GPIB	GPIB[board]::primary_address[::secondary_address]::INSTR
TCPIP	TCPIP[board]::remote_host[::lan_device_name]::INSTR
RSIB	RSIB::remote_host::INSTR
Serial	ASRL[port_number]::INSTR
USB	USB[board]::manid::model_code::serial_No[::interface_No]::INSTR

If you change the `BoardIndex`, `ChassisIndex`, `InterfaceIndex`, `LANName`, `LogicalAddress`, `ManufacturerID`, `ModelCode`, `Port`, `PrimaryAddress`, `RemoteHost`, `SecondaryAddress`, or `SerialNumber` property value, `RsrcName` is automatically updated to reflect the change.

Characteristics

Usage	VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-serial
Read only	Always
Data type	Character vector

Values

The value is defined when the instrument object is created.

Examples

To create a VISA-GPIB object associated with a GPIB controller with board index 0 and an instrument with primary address 1, you supply the following resource name to the `visa` function.

```
vg = visa('ni','GPIB0::1::INSTR');
```

To create a VISA-VXI object associated with a VXI chassis with index 0 and an instrument with logical address 130, you supply the following resource name to the `visa` function.

```
vv = visa('keysight','VXI0::130::INSTR');
```

To create a VISA-GPIB-VXI object associated with a VXI chassis with index 0 and an instrument with logical address 80, you supply the following resource name to the `visa` function.

```
vgv = visa('keysight','GPIB-VXI0::80::INSTR');
```

To create a VISA-serial object associated with the COM1 serial port, you supply the following resource name to the `visa` function.

```
vs = visa('ni','ASRL1::INSTR');
```

See Also

Functions

`visa`

Properties

BoardIndex, ChassisIndex, InterfaceIndex, LANName, LogicalAddress, ManufacturerID, ModelCode, Port, PrimaryAddress, RemoteHost, SecondaryAddress, SerialNumber

SecondaryAddress

Specify secondary address of GPIB instrument

Description

For GPIB and VISA-GPIB objects, you configure `SecondaryAddress` to be the GPIB secondary address associated with your instrument. You can initially specify the secondary address during object creation using the `gpib` or `visa` function. For VISA-GPIB-VXI objects, `SecondaryAddress` is read-only, and the value is returned automatically by the VISA interface after the object is connected to the instrument with the `fopen` function.

For GPIB objects, `SecondaryAddress` can range from 96 to 126, or it can be 0 indicating that no secondary address is used. For VISA-GPIB objects, `SecondaryAddress` can range from 0 to 30. If your instrument does not have a secondary address, then `SecondaryAddress` is 0.

For GPIB and VISA-GPIB objects, the `Name` property is automatically updated to reflect the `SecondaryAddress` value. For VISA-GPIB objects, the `RsrcName` property is automatically updated to reflect the `SecondaryAddress` value.

You can configure `SecondaryAddress` only when the GPIB or VISA-GPIB object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-GPIB-VXI
Read only	While open (GPIB, VISA-GPIB), always (VISA-GPIB-VXI)
Data type	Double

Values

For GPIB objects, `SecondaryAddress` can range from 96 to 126, or it can be 0. For VISA-GPIB objects, `SecondaryAddress` can range from 0 to 30. The default value is `0`.

Examples

This example creates a VISA-GPIB object associated with board 0, primary address 1, and secondary address 8, and then returns the secondary address.

```
vg = visa('keysight','GPIB0::1::8::INSTR');
vg.SecondaryAddress
ans =
     8
```

See Also

Functions

`fclose`, `gpib`, `visa`

Properties

Name, RsrcName, Status

SerialNumber

Specify index of USB instrument on USB hub

Description

You configure `SerialNumber` to be the index of the USB instrument on the USB hub.

The `Name` and `RsrcName` properties are automatically updated to reflect the `SerialNumber` value.

You can configure `SerialNumber` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-USB
Read only	While open
Data type	Character vector

See Also

Functions

`fclose`

Properties

Name, RsrcName

ServerDescription

IVI configuration store server description

Description

`ServerDescription` contains the description of the IVI configuration store server. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Character vector

Sessions

Array of driver sessions in IVI configuration store

Description

`Sessions` identifies the sessions in the IVI configuration store, including the driver sessions.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

See Also

Properties

DriverSessions

Slot

Slot location of VXI instrument

Description

`Slot` indicates the physical slot of the VXI instrument. `Slot` can be any value between `0` and `12`.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	Double

Values

The property value is defined when the instrument object is connected.

SoftwareModules

Array of software modules in IVI configuration store

Description

`SoftwareModules` identifies the software modules in the IVI configuration store. These are installed by the user, but are not configurable. They include instrument drivers.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

Values

The default value is empty when no software modules are installed.

See Also

Properties

DriverSessions, HardwareAssets

SpecificationVersion

IVI configuration server specification version that this server revision complies with

Description

`SpecificationVersion` identifies the specification version of the IVI configuration store server. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Character vector

Status

Status of whether object is connected to instrument

Description

`Status` can be `open` or `closed`. If `Status` is `closed`, the object is not connected to the instrument. If `Status` is `open`, the object is connected to the instrument.

Before you can write or read data, you must connect the object to the instrument with the `fopen` function. You use the `fclose` function to disconnect an object from the instrument.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{closed}`	The object is not connected to the instrument.
`open`	The object is connected to the instrument.

See Also

Functions

`fclose`, `fopen`

StopBits

Specify number of bits used to indicate end of byte

Description

You can configure `StopBits` to be `1`, `1.5`, or `2` for serial port objects, or `1` or `2` for VISA-serial objects If `StopBits` is `1`, one stop bit is used to indicate the end of data transmission. If `StopBits` is `2`, two stop bits are used to indicate the end of data transmission. If `StopBits` is `1.5`, the stop bit is transferred for 150% of the normal time used to transfer one bit.

Note Both the computer and the instrument must be configured to transmit the same number of stop bits.

In addition to the stop bits, the serial data format consists of a start bit, between five and eight data bits, and possibly a parity bit. You specify the number of data bits with the `DataBits` property, and the type of parity checking with the `Parity` property.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	double

Values

Default value is enclosed in braces (`{}`).

Serial Port

`{1}`	One stop bit is transmitted to indicate the end of a byte.
`1.5`	The stop bit is transferred for 150% of the normal time used to transfer one bit.
`2`	Two stop bits are transmitted to indicate the end of a byte.

VISA-Serial

`{1}`	One stop bit is transmitted to indicate the end of a byte.
`2`	Two stop bits are transmitted to indicate the end of a byte

Examples

This example shows how to set the `StopBits` for a serial port object.

Create a serial port object associated with the COM1 port. The default setting for `StopBits` is `1` for serial port objects. Change the value to use two stop bits to indicate the end of data transmission.

```
s = serial('COM1');
s.StopBits = 2;
```

See Also

Properties

DataBits, Parity

Tag

Specify label to associate with instrument object

Description

You configure `Tag` to be a character vector value that uniquely identifies an instrument object.

`Tag` is particularly useful when constructing programs that would otherwise need to define the instrument object as a global variable, or pass the object as an argument between callback routines.

You can return the instrument object with the `instrfind` function by specifying the `Tag` property value.

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Character vector

Values

The default value is an empty character vector.

Examples

Suppose you create a serial port object on a Windows machine associated with the serial port COM1.

```
s = serial('COM1');
fopen(s);
```

You can assign `s` a unique label using `Tag`.

```
s.Tag = 'MySerialObj'
```

You can access `s` in the MATLAB workspace or in a file using the `instrfind` function and the `Tag` property value.

```
s1 = instrfind('Tag','MySerialObj');
```

See Also

Functions

`instrfind`

Terminator

Specify terminator character

Description

For serial, TCPIP, UDP, and VISA-serial objects, you can configure Terminator to an integer value ranging from 0 to 127, to the equivalent ASCII character, or to empty (''). For example, to configure Terminator to a carriage return, you specify the value to be CR or 13. To configure Terminator to a line feed, you specify the value to be LF or 10. For serial port objects, you can also set Terminator to CR/LF or LF/CR. If Terminator is CR/LF, the terminator is a carriage return followed by a line feed. If Terminator is LF/CR, the terminator is a line feed followed by a carriage return. Note that there are no integer equivalents for these two values.

Additionally, you can set Terminator to a 1-by-2 cell array. The first element of the cell is the read terminator and the second element of the cell array is the write terminator.

When performing a write operation using the fprintf function, all occurrences of \n are replaced with the Terminator value. Note that %s\n is the default format for fprintf. A read operation with fgetl, fgets, or fscanf completes when the Terminator value is read. The terminator is ignored for binary operations.

You can also use the terminator to generate a bytes-available event when the BytesAvailableFcnMode is set to terminator.

Characteristics

Usage	Serial, TCPIP, UDP, VISA-serial
Read only	Never
Data type	ASCII value

Values

An integer value ranging from 0 to 127, the equivalent ASCII character, or empty (''). For serial port objects, CR/LF and LF/CR are also accepted values. You specify different read and write terminators as a 1-by-2 cell array.

Examples

This example shows how to set the terminator for a serial port object.

Create a serial port object associated with the COM1 port. The oscilloscope you are connecting to over the serial port is configured to a baud rate of 9600 and a carriage return terminator, so set the serial port object to those values.

```
s = serial('COM1');
s.Baudrate = 9600;
s.Terminator = 'CR';
```

See Also

Functions

`fgetl`, `fgets`, `fprintf`, `fscanf`

Properties

BytesAvailableFcnMode

Timeout

Specify waiting time to complete read or write operation

Description

You configure `Timeout` to be the maximum time (in seconds) to wait to complete a read or write operation.

If a timeout occurs, then the read or write operation aborts. Additionally, if a timeout occurs during an asynchronous read or write operation, then

- An error event is generated.
- The callback function specified for `ErrorFcn` is executed.

Note Timeouts are rounded upwards to full seconds.

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Double

Values

The default value is 10 seconds.

Note that timeouts are rounded upwards to full seconds.

Examples

You can configure the `Timeout` to be the maximum time in seconds to wait to complete a read or write operation for most interfaces.

For example, create a GPIB object `g` associated with a National Instruments GPIB controller with board index `0`, and an instrument with primary address `1`.

```
g = gpib('ni',0,1);
```

You might want to configure the timeout value to a half minute to account for slow data transfer.

```
g.Timeout = 30;
```

Then when you connect to the instrument and do a data read and write, the timeout value of 30 seconds is used.

See Also

Properties

ErrorFcn

TimerFcn

Specify callback function to execute when predefined period passes

Description

You configure `TimerFcn` to execute a callback function when a timer event occurs. A timer event occurs when the time specified by the `TimerPeriod` property passes. Time is measured relative to when the object is connected to the instrument with `fopen`.

Note A timer event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a timer event occurs, the record file records this information:

- The event type as `Timer`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

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

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Callback function

Values

The default value is an empty character vector.

See Also

Functions

`fopen`, `record`

Properties

RecordStatus, TimerPeriod

TimerPeriod

Specify period between timer events

Description

`TimerPeriod` specifies the time, in seconds, that must pass before the callback function specified for `TimerFcn` is called. Time is measured relative to when the object is connected to the instrument with `fopen`.

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

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Double

Values

The default value is 1 second. The minimum value is 0.01 second.

See Also

Functions

`fopen`

Properties

TimerFcn

TransferDelay

Specify use of TCP segment transfer algorithm

Description

You can configure `TransferDelay` to `on` or `off`. If `TransferDelay` is `on`, small segments of outstanding data are collected and sent in a single packet when acknowledgment (ACK) arrives from the server. If `TransferDelay` is `off`, data is sent immediately to the network.

If a network is slow, you can improve its performance by configuring `TransferDelay` to `on`. However, on a fast network acknowledgments arrive quickly and there is negligible difference between configuring `TransferDelay` to `on` or `off`.

Note that the segment transfer algorithm used by `TransferDelay` is Nagle's algorithm.

Characteristics

Usage	TCPIP
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{on}`	Use the TCP segment transfer algorithm.
`off`	Do not use the TCP segment transfer algorithm.

See Also

Functions

`tcpip`

TransferStatus

Status of whether asynchronous read or write operation is in progress

Description

`TransferStatus` can be `idle`, `read`, `write`, or `read&write`. If `TransferStatus` is `idle`, then no asynchronous read or write operations are in progress. If `TransferStatus` is `read`, then an asynchronous read operation is in progress. If `TransferStatus` is `write`, then an asynchronous write operation is in progress. If `TransferStatus` is `read&write`, then both an asynchronous read and an asynchronous write operation are in progress.

You can write data asynchronously using the `fprintf` or `fwrite` functions. You can read data asynchronously using the `readasync` function, or by configuring `ReadAsyncMode` to `continuous` (serial, TCPIP, UDP, and VISA-serial objects only). For detailed information about asynchronous read and write operations, refer to “Communicating with Your Instrument” on page 2-8.

While `readasync` is executing for any instrument object, `TransferStatus` might indicate that data is being read even though data is not filling the input buffer. However, if `ReadAsyncMode` is `continuous`, `TransferStatus` indicates that data is being read only when data is actually filling the input buffer.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{idle}`	No asynchronous operations are in progress.
`read`	An asynchronous read operation is in progress.
`write`	An asynchronous write operation is in progress.
`read&write`	Asynchronous read and write operations are in progress.

See Also

Functions

`fprintf`, `fwrite`, `readasync`

Properties

ReadAsyncMode

TriggerFcn

Specify callback function to execute when trigger event occurs

Description

You configure `TriggerFcn` to execute a callback function when a trigger event occurs. A trigger event is generated when a trigger occurs in software, or on one of the VXI hardware trigger lines. You configure the trigger type with the `TriggerType` property.

Note A trigger event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a trigger event occurs, the record file records

- The event type as `Trigger`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Characteristics

Usage	VISA-VXI
Read only	Never
Data type	Character vector

Values

The default value is an empty character vector.

See Also

Functions

`record`

Properties

RecordStatus, TriggerLine, TriggerType

TriggerLine

Specify trigger line on VXI instrument

Description

You can configure `TriggerLine` to be `TTL0` through `TTL7`, `ECL0`, or `ECL1`. You can use only one trigger line at a time.

You can specify the trigger type with the `TriggerType` property. When `TriggerType` is `hardware`, the line triggered is given by the `TriggerLine` value. When the `TriggerType` is `software`, the `TriggerLine` value is ignored.

You execute a trigger for a VISA-VXI object with the `trigger` function.

Characteristics

Usage	VISA-VXI
Read only	Never
Data type	Character vector

Values

`TriggerLine` can be `TTL0` through `TTL7`, `ECL0`, or `ECL1`. The default value is `TTL0`.

See Also

Functions

`trigger`

Properties

TriggerType

TriggerType

Specify trigger type

Description

You can configure `TriggerType` to be `software` or `hardware`. If `TriggerType` is `software`, then a software trigger is used. If `TriggerType` is `hardware`, then the trigger line specified by the `TriggerLine` property is used.

You execute a trigger for a VISA-VXI object with the `trigger` function.

Characteristics

Usage	VISA-VXI
Read only	Never
Data type	Character vector

Values

Default value is enclosed in braces (`{}`).

`{hardware}`	A hardware trigger is used.
`software`	A software trigger is used.

See Also

Functions

`trigger`

Properties

TriggerLine

Type

Instrument object type

Description

`Type` indicates the type of the object. `Type` is automatically defined after the instrument object is created with the `serial`, `gpib`, or `visa` function.

Using the `instrfind` function and the `Type` value, you can quickly identify instrument objects of a given type.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Character vector

Values

`gpib`	The object type is GPIB.
`serial`	The object type is serial port.
`tcpip`	The object type is TCPIP.
`udp`	The object type is UDP.
`visa-gpib`	The object type is VISA-GPIB.
`visa-vxi`	The object type is VISA-VXI.
`visa-gpib-vxi`	The object type is VISA-GPIB-VXI.
`visa-serial`	The object type is VISA-serial.

The value is automatically determined when the instrument object is created.

Examples

Create a serial port object on a Windows machine associated with the serial port COM1. The value of the `Type` property is `serial`, which is the object class.

```
s = serial('COM1');
s.Type
ans =
serial
```

See Also

Functions

`instrfind`, `gpib`, `serial`, `tcpip`, `udp`, `visa`

UserData

Specify data to associate with instrument object

Description

You configure `UserData` to store data that you want to associate with an instrument object. The object does not use this data directly, but you can access it using dot notation.

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Any type

Values

The default value is an empty vector.

Examples

Create the serial port object on a Windows machine associated with the serial port COM1.

```
s = serial('COM1');
```

You can associate data with `s` by storing it in `UserData`.

```
coeff.a = 1.0;
coeff.b = -1.25;
s.UserData = coeff
```

ValuesReceived

Total number of values read from instrument

Description

`ValuesReceived` indicates the total number of values read from the instrument. The value is updated after each successful read operation, and is set to `0` after the `fopen` function is issued. If the terminator is read from the instrument, then this value is reflected by `ValuesReceived`.

If you are reading data asynchronously, use the `BytesAvailable` property to return the number of bytes currently available in the input buffer.

When performing a read operation, the received data is represented by values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Double

Values

The default value is `0`.

Examples

Suppose you create a serial port object on a Windows machine associated with the serial port COM1.

```
s = serial('COM1');
fopen(s)
```

If you write the `RS232?` command, and then read back the response using `fscanf`, `ValuesReceived` is `17` because the instrument is configured to send the `LF` terminator.

```
fprintf(s,'RS232?')
out = fscanf(s)
out =
9600;0;0;NONE;LF
s.ValuesReceived
ans =
    17
```

See Also

Functions

`fopen`

Properties

BytesAvailable

ValuesSent

Total number of values written to instrument

Description

ValuesSent indicates the total number of values written to the instrument. The value is updated after each successful write operation, and is set to 0 after the fopen function is issued. If you are writing the terminator, then ValuesSent reflects this value.

If you are writing data asynchronously, use the BytesToOutput property to return the number of bytes currently in the output buffer.

When performing a write operation, the transmitted data is represented by values rather than bytes. A value consists of one or more bytes. For example, one uint32 value consists of four bytes.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Double

Values

The default value is 0.

Examples

Create a serial port object on a Windows machine associated with the serial port COM1.

```
s = serial('COM1');
fopen(s)
```

If you write the *IDN? command using the fprintf function, then ValuesSent is 6 because the default data format is %s\n, and the terminator was written.

```
fprintf(s,'*IDN?')
s.ValuesSent
ans =
    6
```

See Also

Functions

fopen

Properties

BytesToOutput

Vendor

IVI configuration server vendor

Description

`Vendor` identifies the vendor of the IVI configuration server. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Character vector

visadev Properties

Access VISA resource properties

Description

Configure your VISA resource and its communication settings using its properties. After you create a `visadev` object, you can use dot notation to read and set properties.

Properties

Object Creation Properties

`ResourceName` — VISA resource name
string scalar

VISA resource name, returned as a string scalar. Identify the resource name of the device you want to connect to using the information returned by `visadevlist`. This property can be set only at object creation.

Each type of VISA interface has a different format, described in the following table. The VISA resource name format and its parameters are defined by the VISA standard specifications. Replace the *italicized* text with values for the specified parameters. The parameters in brackets are optional.

Interface	Resource Name
TCP/IP (using VXI-11 or HiSLIP)	TCPIP[*board*]::*remote_host*[::*lan_device_name*]::INSTR
TCP/IP Socket	TCPIP[*board*]::*remote_host*::*port*::SOCKET
USB	USB[*board*]::*vendor_ID*::*product_ID*::*serial_number*[::*interface_number*]::INSTR
GPIB	GPIB[*board*]::*primary_address*[::*secondary_address*]::INSTR
Serial	ASRL[*port_number*]::INSTR
VXI	VXI[*chassis*]::*VXI_logical_address*::INSTR
PXI	PXI[*bus*]::*device*[::*function*][::INSTR] PXI[*bus*]::CHASSIS*chassis*::SLOT*slot*[::FUNC*function*][::INSTR]

The resource name parameters are described as follows. Each parameter corresponds to one of the `visadev` properties.

Interface	Parameter	Description
All	*board*	Board index (default value of 0)
TCP/IP (using VXI-11 or HiSLIP) and TCP/IP Socket	*remote_host*	Remote host name or IP address of the instrument
	lan_device_name	Local Area Network (LAN) device name (default value of inst0)

Interface	Parameter	Description
	port	Remote host port for TCP/IP socket
USB	*vendor_ID*	Manufacturer ID of the USB instrument
	product_ID	Model code for the USB instrument
	serial_number	Index of the instrument on the USB hub
	interface_number	USB interface
GPIB	*primary_address*	Primary address of the GPIB instrument
	secondary_address	Secondary address of the GPIB instrument (default value of 0)
Serial	*port_number*	Serial port number (default value of 1)
VXI and PXI	*chassis*	VXI or PXI chassis index (default value of 0 for VXI)
	VXI_logical_address	Logical address of the VXI instrument
	bus	PCI bus number
	device	PCI device number
	function	PCI function number (default value of 0)
	slot	Slot number

Example: `gpibdev = visadev("GPIB0::5::INSTR")` connects to the GPIB device specified by the VISA resource name `GPIB0::5::INSTR`.

Data Types: `char` | `string`

`Alias` — VISA alias associated with resource
string scalar

VISA alias associated with resource, returned as a string scalar. Identify the alias of the device you want to connect to using the information returned by `visadevlist`. The alias is defined in your VISA vendor's configuration utility software. This property can be set only at object creation.

Example: `serialdev = visadev("COM4")` connects to the serial device specified by the VISA resource alias `COM4`.

Data Types: `char` | `string`

Type — Type of VISA resource
`gpib` | `pxi` | `serial` | `socket` | `tcpip` | `usb` | `vxi`

This property is read-only.

Type of VISA resource, returned as one of the supported VISA interfaces. Some properties and object functions are specific to an interface type.

Example: `v.Type` returns the type of VISA resource.

`Vendor` — Instrument manufacturer
string scalar

This property is read-only.

Instrument manufacturer, returned as a character vector or string scalar. This property is empty if the VISA interface type does not provide information about the manufacturer.

Example: `v.Vendor` returns the name of the instrument manufacturer.

Data Types: `string`

`Model` — Instrument model
string scalar

This property is read-only.

Instrument model, returned as a character vector or string scalar. This property is empty if the VISA interface type does not provide information about the model.

Example: `v.Model` returns the name of the instrument model.

Data Types: `string`

`SerialNumber` — Unique serial number associated with instrument
string scalar

This property is read-only.

Unique serial number associated with instrument, returned as a character vector or string scalar. This property is empty if the VISA interface type does not provide information about the serial number.

Example: `v.SerialNumber` returns the instrument serial number.

Data Types: `string`

Read and Write Properties

`ByteOrder` — Sequential order of bytes
`"little-endian"` (default) | `"big-endian"`

Sequential order in which bytes are arranged into larger numerical values, returned as `"little-endian"` or `"big-endian"`.

Example: `v.ByteOrder = "big-endian"` sets the byte order to big-endian.

Data Types: `char` | `string`

`Timeout` — Allowed time to complete operations
10 (default) | numeric

Allowed time in seconds to complete read and write operations, returned as a numeric value.

Example: `v.Timeout = 20` sets the timeout period to 20 seconds.

Data Types: `double`

`Terminator` — Terminator character for data
`"LF"` (default) | `"CR"` | `"CR/LF"` | 0 to 255

Terminator character for reading and writing ASCII-terminated data, returned as `"LF"`, `"CR"`, `"CR/LF"`, or a number from 0 to 255, inclusive. If the read and write terminators are different,

`Terminator` is returned as a 1x2 cell array of these values. Set this property with the `configureTerminator` function.

Example: `configureTerminator(v,"CR")` sets both the read and write terminators to `"CR"`.

Example: `configureTerminator(v,"CR",10)` sets the read terminator to `"CR"` and the write terminator to `10`.

Data Types: `double` | `char` | `string`

`NumBytesAvailable` — Number of bytes available to read
0 (default) | numeric

This property is read-only.

Number of bytes available to read, returned as a numeric value.

Example: `v.NumBytesAvailable` returns the number of bytes available to read.

Data Types: `double`

`NumBytesWritten` — Total number of bytes written
0 (default) | numeric

This property is read-only.

Total number of bytes written, returned as a numeric value.

Example: `v.NumBytesWritten` returns the number of bytes written.

Data Types: `double`

Callback Properties

`BytesAvailableFcnMode` — Bytes available callback trigger mode
`"off"` (default) | `"byte"` | `"terminator"`

Bytes available callback trigger mode, returned as `"off"`, `"byte"`, or `"terminator"`. This setting determines if the callback is off, triggered by the number of bytes specified by `BytesAvailableFcnCount`, or triggered by the terminator specified by `Terminator`. Set this property with the `configureCallback` function.

Example: `configureCallback(v,"byte",50,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 50 bytes of new data are available to be read.

Example: `configureCallback(v,"terminator",@callbackFcn)` sets the `callbackFcn` callback to trigger when a terminator is available to be read.

Example: `configureCallback(v,"off")` turns off callbacks.

Data Types: `char` | `string`

`BytesAvailableFcnCount` — Number of bytes of data to trigger callback
64 (default) | numeric

Number of bytes of data to trigger the callback specified by `BytesAvailableFcn`, returned as a double. This value is used only when the `BytesAvailableFcnMode` property is `"byte"`. Set these properties with the `configureCallback` function.

Example: `configureCallback(v,"byte",50,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 50 bytes of new data are available to be read.

Data Types: `double`

`BytesAvailableFcn` — Callback function triggered by bytes available event
function handle

Callback function triggered by a bytes available event, returned as a function handle. A bytes available event is generated by receiving a certain number of bytes or a terminator. This property is empty until you assign a function handle. Set this property with the `configureCallback` function.

Example: `configureCallback(v,"byte",50,@callbackFcn)` sets the `callbackFcn` callback to trigger each time 50 bytes of new data are available to be read.

Data Types: `function_handle`

`ErrorOccurredFcn` — Callback function triggered by error event
function handle

Callback function triggered by an error event, returned as a function handle. An error event is generated when the connection to your VISA resource is interrupted or when an asynchronous read error occurs. This property is empty until you assign a function handle.

Example: `v.ErrorOccurredFcn = @myErrorFcn`

Data Types: `function_handle`

`UserData` — General purpose property for user data
any type

General purpose property for user data, returned as any MATLAB data type. For example, you can use this property to store data when an event is triggered from a callback function.

Example: `v.UserData`

VISA-TCP/IP

`LANName` — LAN device name
string scalar

This property is read-only.

LAN device name, returned as a string.

Example: `v.LANName` returns the LAN device name.

Data Types: `string`

`InstrumentAddress` — TCP/IP address of instrument
string scalar

This property is read-only.

TCP/IP address of instrument in dot-decimal notation, returned as a string.

Example: `v.InstrumentAddress` returns the IP address of the instrument.

Data Types: `string`

`BoardIndex` — Index number of network board associated with instrument
numeric

This property is read-only.

Index number of network board associated with instrument, returned as a positive integer value.

Example: `v.BoardIndex` returns the network board index number.

Data Types: `double`

VISA-Socket

`IPAddress` — TCP/IP address of socket
string scalar

This property is read-only.

TCP/IP address of socket in dot-decimal notation, returned as a string.

Example: `v.IPAddress` returns the IP address of the socket.

Data Types: `string`

`Port` — Port number for given TCP/IP address
string scalar

This property is read-only.

Port number for given TCP/IP address, returned as a string.

Example: `v.Port` returns the port number associated with the TCP/IP address.

Data Types: `string`

VISA-USB Properties

`VendorID` — Manufacturer ID number of device
string scalar

This property is read-only.

Manufacturer ID number of device (VID), returned as a string.

Example: `v.VendorID` returns the vendor ID.

Data Types: `string`

`ProductID` — Model code of device
string scalar

This property is read-only.

Model code of device (PID), returned as a string.

Example: `v.ProductID` returns the product ID.

Data Types: `string`

`BoardIndex` — USB board number
numeric

This property is read-only.

USB board number, returned as a positive integer value.

Example: `v.BoardIndex` returns the USB board number.

Data Types: `double`

`InterfaceIndex` — USB interface number
numeric

This property is read-only.

USB interface number, returned as a positive integer value.

Example: `v.InterfaceIndex` returns the USB interface number.

Data Types: `double`

VISA-GPIB Properties

`BoardIndex` — GPIB board index
numeric

This property is read-only.

GPIB board index, returned as a positive integer value.

Example: `v.BoardIndex` returns the GPIB board index.

Data Types: `double`

`PrimaryAddress` — GPIB primary address
numeric

This property is read-only.

GPIB primary address associated with instrument, returned as an integer from 0 to 30, inclusive.

Example: `v.PrimaryAddress` returns the GPIB primary address.

Data Types: `double`

`SecondaryAddress` — GPIB secondary address
numeric

This property is read-only.

GPIB secondary address associated with instrument, returned as an integer from 0 to 30, inclusive.

Example: `v.SecondaryAddress` returns the GPIB secondary address.

Data Types: `double`

`EOIMode` — EOI Mode
`"on"` (default) | `"off"`

EOI mode, returned as `on` or `off`. This property specifies whether the EOI (end or identify) line is asserted at the end of a write operation.

Example: `v.EOIMode = "off"` does not assert an EOI line at the end of a write.

Data Types: `char` | `string`

VISA-Serial

Port — Serial communication port
string scalar

This property is read-only.

Serial communication port, returned as a string.

Example: `v.Port` returns the serial communication port.

Data Types: `string`

BaudRate — Communication speed
9600 (default) | double

Speed of serial communication in bits per second, returned as a positive integer.

Example: `v.BaudRate = 14400` sets the baud rate to 14400.

Data Types: `double`

DataBits — Number of bits to represent one character of data
8 (default) | 7 | 6 | 5

Number of bits to represent one character of data, returned as 8, 7, 6, or 5.

Example: `v.DataBits = 5` sets the number of data bits to 5 bits.

Data Types: `double`

StopBits — Pattern of bits that indicates the end of a character
1 (default) | 1.5 | 2

Pattern of bits that indicates the end of a character or of the whole transmission, returned as 1, 1.5, or 2. This property depends on the value of the `DataBits` property as follows.

Value of DataBits	Supported Values of StopBits
8, 7, or 6	1 and 2
5	1 and 1.5

Example: `v.StopBits = 2` sets the number of stop bits to 2.

Data Types: `double`

Parity — Parity bit type
"none" (default) | "even" | "odd"

Parity bit type added to data transmitted by serial port, returned as "none", "even", or "odd". You can use this property to add a parity bit (also referred to as a check bit) to your data. Adding a parity bit to a string of binary code is a method of detecting errors in data transmission by ensuring that the total number of 1-bits is even or odd.

The value of the parity bit is determined by the number of 1s in a given set of bits and is set as follows.

Parity Bit Type	Parity Bit Value	
	If number of 1s is even	If number of 1s is odd
none	No parity bit set	No parity bit set
even	0	1
odd	1	0

Example: `v.Parity = "even"` sets the parity bit type to even.

Data Types: `char` | `string`

`FlowControl` — Mode for managing data transmission rate
`"none"` (default) | `"hardware"` | `"software"`

Mode for managing data transmission rate, returned as `"none"`, `"hardware"`, or `"software"`. Specify `"none"` to have no flow control, `"hardware"` to let your hardware determine the flow control, and `"software"` to let your software determine the flow control.

Example: `v.FlowControl = "hardware"` sets the flow control to hardware.

Data Types: `char` | `string`

VISA-VXI and VISA-PXI Properties

Bus — PCI bus number
numeric

This property is read-only.

PCI bus number for device, returned as a positive number. This property is only for VISA-PXI interface objects.

Example: `v.Bus` returns the device's PCI bus number.

Data Types: `double`

`DeviceIndex` — PXI device number
numeric

This property is read-only.

PXI device number, returned as a positive number. This property is only for VISA-PXI interface objects.

Example: `v.DeviceIndex` returns the PXI device number.

Data Types: `double`

`FunctionIndex` — PXI function number
numeric

This property is read-only.

PXI function number, returned as a positive number. This property is only for VISA-PXI interface objects.

Example: `v.FunctionIndex` returns the PXI function number.

Data Types: `double`

`ChassisIndex` — PXI or VXI chassis index number
numeric

This property is read-only.

PXI or VXI chassis index number, returned as a positive number.

Example: `v.ChassisIndex` returns the PXI or VXI chassis index number.

Data Types: `double`

`LogicalAddress` — VXI instrument logical address
numeric

This property is read-only.

VXI instrument logical address, returned as a positive number. This property is only for VISA-VXI interface objects.

Example: `v.LogicalAddress` returns the VXI instrument logical address.

Data Types: `double`

`Slot` — PXI or VXI instrument slot location
numeric

This property is read-only.

PXI or VXI instrument slot location, returned as a positive number.

Example: `v.Slot` returns the PXI or VXI instrument slot location.

Data Types: `double`

`EOIMode` — EOI Mode
`"on"` (default) | `"off"`

EOI mode, returned as `on` or `off`. This property specifies whether the EOI (end or identify) line is asserted at the end of a write operation.

Example: `v.EOIMode = "off"` does not assert an EOI line at the end of a write.

Data Types: `char` | `string`

See Also

`visadevlist` | `visadev` | `configureTerminator` | `configureCallback`

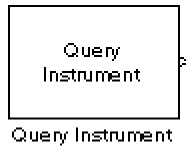
External Websites
IVI Specifications

Introduced in R2021a

Blocks

Query Instrument

Query or read instrument data



Library

Instrument Control Toolbox

Description

The Query Instrument block configures and opens an interface to an instrument, initializes the instrument, and queries the instrument for data. The configuration and initialization happen at the start of the model execution. The block queries the instrument for data during model run time.

The block has no input ports. The block has one output port corresponding to the data received from the instrument.

Other Supported Features

- This block supports the use of Simulink Accelerator™ mode, but not Rapid Accelerator or code generation.
- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the “Simulink” documentation.

Parameters

Block sample time

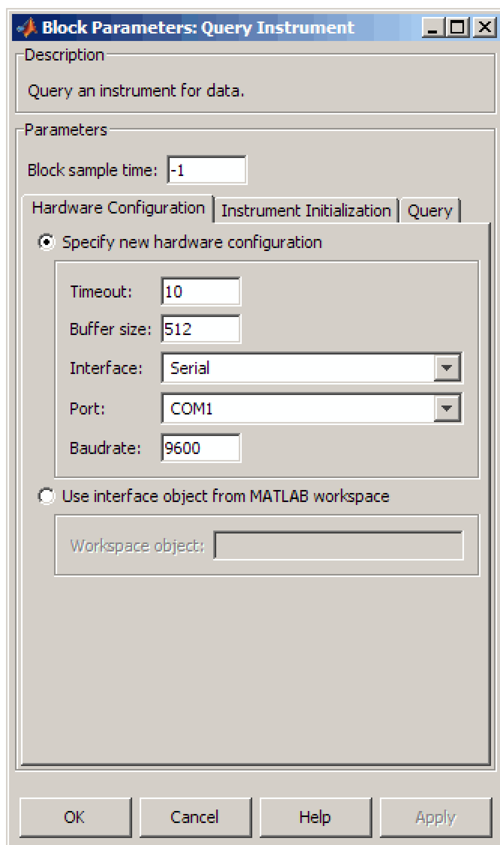
The Block sample time parameter is the only setting outside of the dialog box tabs. The default value is `1`. Setting the value to `-1` sets the block to inherit timing. A positive value is used as the sample period.

Hardware Configuration Tab

The **Hardware Configuration** tab is where you define the settings for communications with your instrument. You have two choices about establishing an interface:

- Specify a new hardware configuration.
- Use an interface object from the MATLAB workspace.

The following figure shows the **Hardware Configuration** tab set to specify a new hardware configuration using a serial port interface.



Because some parameters apply to multiple interface types, they appear here in alphabetical order.

Baudrate

The rate at which bits are transmitted for the serial or VISA serial interface.

Board index

The index of the board used for GPIB, VISA GPIB, VISA TCPIP, or VISA USB interface to the instrument. See BoardIndex property for more information.

Board vendor

The vendor of the GPIB board used for the interface to the instrument. Your choices are Keysight (formerly Agilent), ICS Electronics, Measurement Computing (MCC), and National Instruments.

Chassis index

The index number of the VXI chassis. Used for VISA VXI and VISA VXI-GPIB interface types.

Buffer size

The total number of bytes that can be stored in the software input buffer during a read operation.

Interface

Select the type of hardware interface to the instrument. Your options are those interfaces supported by the Instrument Control Toolbox software. The previous figure shows a configuration for a serial port interface.

Logical address

The logical address of the VXI instrument. Used for VISA VXI and VISA VXI-GPIB interface types.

Manufacturer ID

The manufacturer ID of the VISA USB instrument. See ManufacturerID property for more information.

Model code

The model code of the VISA USB instrument. See ModelCode property for more information.

Port

The port for the serial interface: COM1, COM2, etc.

Primary address

The primary address of the instrument on the GPIB.

Remote host

The host name or IP address of the instrument. Used for UDP, TCPIP, or VISA TCPIP interface types.

Remote port

The port on the instrument or remote host used for communication. Used for UDP, TCPIP, or VISA TCPIP interface types.

Secondary address

The secondary address of the instrument on the GPIB.

Serial number

The serial number of the VISA USB instrument defined as a character vector. See SerialNumber property for more information.

Timeout

Time in seconds allowed to complete the query operation.

VISA vendor

The vendor of the VISA used for any of the VISA interface types. Your choices are Keysight (formerly Agilent), National Instruments, and Tektronix.

Use interface object from MATLAB workspace

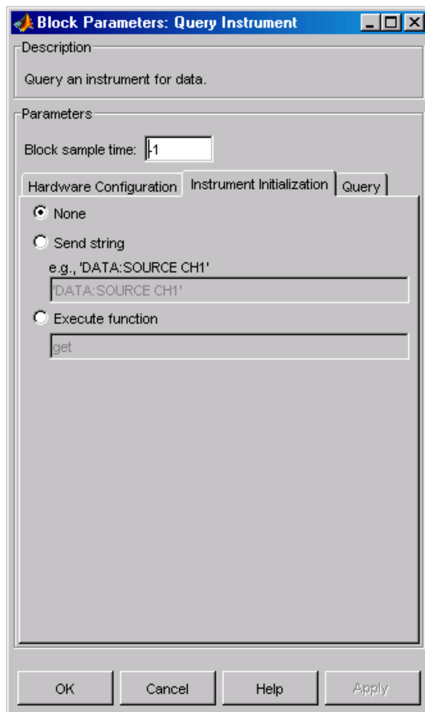
Select this option to use an interface object from the MATLAB workspace.

Workspace object

Enter the object name that you want to use from the MATLAB workspace.

Instrument Initialization Tab

The **Instrument Initialization** tab is where you define what happens when you first open your connection to the instrument.



None

The default initialization option is none.

Send string

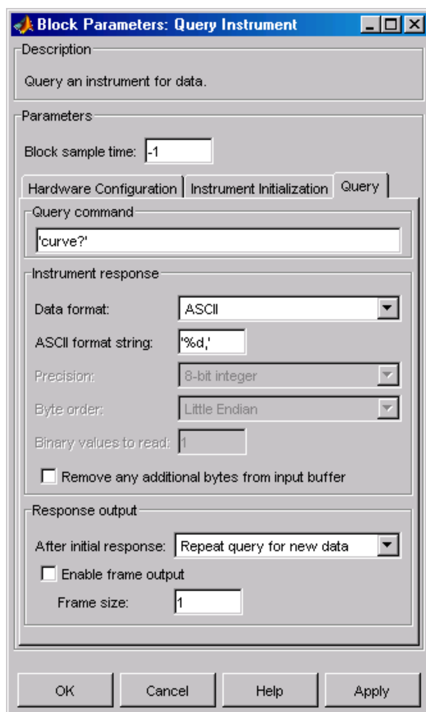
A string sent to the instrument as an instrument command to initialize the instrument or set it up in a known state.

Execute function

Any function that has as its only argument the interface object representing the instrument. You can write this function to include several instrument commands and initialization data.

Query Tab

The **Query** tab is where you define the optional query command, set the format for the response, and define what the block does after the initial instrument response.



Query command

This is the query command that is sent to the instrument. It is usually a request for instrument status or data. *This command is optional* — if you are retrieving information or data from the instrument and no query command is necessary to do that, you can leave this field blank.

Data format

Your options are `ASCII`, `Binary`, or `Binblock` (binary block — the `binblock` format is described in the `binblockwrite` function reference page).

ASCII format string

Available only when the format is `ASCII`, this defines the format string for the data. For a list of formats, see the `fscanf` function.

Precision

Used for binary or binblock format. Your options are:

- `8-bit integer` (default)
- `16-bit integer`
- `32-bit integer`
- `8-bit unsigned integer`
- `16-bit unsigned integer`
- `32-bit unsigned integer`
- `32-bit float`
- `64-bit float`

Byte order

When using binary or binblock format with more than 8 bits, you can specify the instrument's byte order for the data. Your options are `Big Endian` or `Little Endian`.

Binary values to read

Used for binary format. Specify the number of binary values to read from the instrument.

Remove additional bytes from input buffer

Select this option if you want to remove any additional bytes from the input buffer before querying.

After initial response

This defines the action to take after the first response from the instrument. Your options are Repeat query for new data, Recycle original data, Hold final value, Output zero, or Stop simulation.

Enable frame output

A frame is a sequence of samples combined into a single vector. In frame-based processing all the samples in a frame are processed simultaneously. In sample-based processing, samples are processed one at a time. The advantage of frame-based processing is that it can greatly increase the speed of a simulation. For example, you might use frames if you are reading a waveform from your instrument rather than a single-point measurement.

Frame size

Frame size determines the number of samples in a frame.

Note Hardware information shown in the dialog box is determined and cached when you first open the dialog box. To refresh the display with new values, restart MATLAB.

See Also

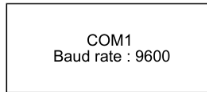
Serial Configuration, Serial Receive, Serial Send, TCP/IP Receive, TCP/IP Send, To Instrument, UDP Receive, UDP Send

Introduced before R2006a

Serial Configuration

Configure parameters for serial port

Library: Instrument Control Toolbox



Description

The Serial Configuration block configures parameters for a serial port that you can use to send and receive data. Use this block to set the parameters of your serial port before you set up the Serial Receive and the Serial Send blocks.

Note You must configure your serial port parameters using the Serial Configuration block before you specify the Serial Receive and Serial Send block parameters.

Other Supported Features

- The Serial Configuration block supports the use of Simulink Accelerator mode, but not Rapid Accelerator. This feature speeds up the execution of Simulink models.
- The Serial Configuration block supports the use of model referencing. This feature lets your model include other Simulink models as modular components.
- The Serial Configuration block supports C/C++ code generation. This feature allows you to generate C and C++ code using Simulink Coder™.

For more information on these features, see the “Simulink” documentation.

Parameters

`Port` — Serial communication port

available communication ports

Serial port on your machine that you want to configure. Use this configured port to send and receive data with your Serial Send and Serial Receive blocks. If you have not configured a port, the block returns an error when you run your model.

Note Each Serial Send and Serial Receive block must have a configured Serial Configuration block. If you use multiple serial ports in your simulation, you must configure each port using a separate Serial Configuration block.

Programmatic Use

Block Parameter: `Port`
Type: character vector, string

`Baud rate` — Communication speed

9600 (default) | positive integer

Rate at which bits are transmitted for the serial interface, in bits per second.

Programmatic Use
Block Parameter: `BaudRate`
Type: character vector, string
Values: positive integer
Default: `'9600'`

`Data bits` — Number of bits to represent one character of data

8 (default) | 5 | 6 | 7

Number of data bits to transmit over the serial interface.

Programmatic Use
Block Parameter: `DataBits`
Type: character vector, string
Values: `'5'` | `'6'` | `'7'` | `'8'`
Default: `'8'`

`Parity` — Parity bit type

none (default) | even | odd

Parity bit type added to data transmitted by serial port. You can use this parameter to add a parity bit (also referred to as a check bit) to your data. Adding a parity bit to a string of binary code is a method of detecting errors in data transmission by ensuring that the total number of 1-bits is even or odd.

The value of the parity bit is determined by the number of 1s in a given set of bits and is set as follows.

Parity Bit Type	Parity Bit Value	
	If number of 1s is even	If number of 1s is odd
`none`	No parity bit set	No parity bit set
`even`	0	1
`odd`	1	0

Note Starting in R2021a, the **Parity** parameter no longer supports `mark` or `space`. For more information, see “Compatibility Considerations” on page 26-10.

Programmatic Use
Block Parameter: `Parity`
Type: character vector, string
Values: `'none'` | `'even'` | `'odd'`
Default: `'none'`

`Stop bits` — Pattern of bits that indicates the end of a character

1 (default) | positive scalar

Number of bits used to indicate the end of a byte. This parameter depends on the value you select for the **Data bits** parameter. If you select data bits `6`, `7`, or `8`, the default value is `1` and the other available choice is `2`. If you select data bit `5`, the default value is `1` and the other available choice is `1.5`.

Programmatic Use
Block Parameter: `StopBits`
Type: character vector, string
Values: positive scalar
Default: `'1'`

`Byte order` — Sequential order of bytes

`little-endian` (default) | `big-endian`

Sequential order in which bytes are arranged into larger numerical values. If the byte order is `little-endian`, then the instrument stores the first byte in the first memory address. If the byte order is `big-endian`, then the instrument stores the last byte in the first memory address.

Configure the byte order to the appropriate value for your instrument before performing a read or write operation. Refer to your instrument documentation for information about the order in which it stores bytes.

Programmatic Use
Block Parameter: `ByteOrder`
Type: character vector, string
Values: `'little-endian'` | `'big-endian'`
Default: `'little-endian'`

`Flow control` — Mode for managing data transmission rate

`none` (default) | `hardware`

Process of managing the rate of data transmission on your serial port. Select `none` to have no flow control or `hardware` to let your hardware determine the flow control.

Programmatic Use
Block Parameter: `FlowControl`
Type: character vector, string
Values: `'none'` | `'hardware'`
Default: `none`

`Timeout` — Allowed time to complete operations

10 (default) | positive scalar

Amount of time that the model waits for data during each simulation time step.

Programmatic Use
Block Parameter: `Timeout`
Type: character vector, string
Values: positive scalar
Default: `'10'`

Compatibility Considerations

Parity parameter no longer supports `mark` or `space` in Serial Configuration block

Errors starting in R2021a

The `mark` and `space` options for the **Parity** parameter are no longer supported in the Serial Configuration block. Valid values for **Parity** are `none` (default), `even`, and `odd`.

If you try to open an existing model that has the **Parity** value set to `mark` or `space`, MATLAB returns a warning and changes the parameter to the default value `none`.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates platform-specific code for the host machine's platform only (Windows, macOS, Linux).

See Also
Serial Receive | Serial Send

Introduced in R2008a

Serial Receive

Receive binary data over serial port

Library: Instrument Control Toolbox

Description

The Serial Receive block configures and opens an interface to the specified serial port. The configuration and initialization occur once at the start of the model's execution. The block acquires data from the serial port during the model's run time. You can use only one Serial Receive block at a time to receive data from a specific serial port.

Note You must configure your serial port parameters using the Serial Configuration block before you specify the Serial Receive block parameters.

This block has no input ports. It has one or two output ports based on whether you select blocking or non-blocking mode. If you select blocking mode, the block has one output port, **Data**, corresponding to the data it receives. If you do not select blocking mode, the block has two output ports, **Data** and **Status**.

This block uses a First In, First Out (FIFO) buffer to receive data from the serial port. At each time step, the **Data** port returns the requested values from the buffer. In non-blocking mode, the **Status** port indicates if the block has received new data. If the **Status** port displays `1`, new data is available and if the **Status** port displays `0`, no new data is available.

Other Supported Features

- The Serial Receive block supports the use of Simulink Accelerator mode, but not Rapid Accelerator. This feature speeds up the execution of Simulink models.
- The Serial Receive block supports the use of model referencing. This feature lets your model include other Simulink models as modular components.
- The Serial Receive block supports C/C++ code generation. This feature allows you to generate C and C++ code using Simulink Coder.

For more information on these features, see the “Simulink” documentation.

Ports

Output

`Data` — Data received

vector | matrix | array

Data received by the block from the serial port, returned as a vector, matrix, or array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

`Status` — New data available
`false` or `0` (default) | `true` or `1`

New data available status, returned as numeric or logical `1` (`true`) or `0` (`false`). If this port returns `1`, new data is available to be read.

Dependencies

To enable this port, unselect the **Enable blocking mode** parameter.

Data Types: `Boolean`

Parameters

`Port` — Serial communication port
available communication ports

Serial port on your machine that you want to receive data from. Select a port from the available ports and then configure the port using the Serial Configuration block. If you have not configured a port, the block returns an error when you run your model.

Note Each Serial Receive block must have a configured Serial Configuration block. If you use multiple serial ports in your simulation, you must configure each port using a separate Serial Configuration block.

Programmatic Use
Block Parameter: `Port`
Type: character vector, string

`Data type` — Output data type
`uint8` (default) | `single` | `double` | `int8` | `int16` | `uint16` | `int32` | `uint32`

Data type that the block receives from the serial port, specified as a MATLAB numeric data type.

Programmatic Use
Block Parameter: `DataType`
Type: character vector, string
Values: `'uint8'` | `'single'` | `'double'` | `'int8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'`
Default: `'uint8'`

`Header` — Header
numeric array | integer from 0 to 255, inclusive

If this parameter is selected, you can specify the header that indicates the beginning of your data block. The simulation disregards data that occurs before the header. The header data is not sent to the output port. By default, this parameter is not selected and no header is specified.

The numeric array specified in this parameter is the `uint8` integer representation of the corresponding ASCII characters. The exact form of this parameter depends on the type of the ASCII character.

Type of ASCII Character	Example ASCII Character	MATLAB Command	Parameter Value
Special character	"#"	uint8('#')	[35]
Numeric	"81"	uint8('81')	[56 49]
Alphabet	"Start"	uint8('Start')	[83 116 97 114 116]

You can also specify this parameter using the hexadecimal representation of the ASCII characters.

Programmatic Use
Block Parameter: ToggleHeader
Type: character vector, string
Values: 'on' | 'off'
Default: 'off'
Block Parameter: Header
Type: character vector, string
Values: integer array

Terminator — Terminator

<none> (default) | CR ('\r') | LF ('\n') | CR/LF ('\r\n') | NULL ('\0') | Custom Terminator

If this parameter is selected, you can specify the terminator that indicates the end of your data block. The simulation considers any data that occurs after the terminator as a new data block. The terminator data is not sent to the output port. This terminator must match the terminator in the data you are reading from your serial port, if it has one.

If you select Custom Terminator, you can specify your own terminator value.

Programmatic Use
Block Parameter: ToggleTerminator
Type: character vector, string
Values: 'on' | 'off'
Default: 'off'
Block Parameter: Terminator
Type: character vector, string
Values: '<none>' | 'CR ('\r')' | 'LF ('\n')' | 'CR/LF ('\r\n')' | 'NULL ('\0')' | 'Custom Terminator'
Default: '<none>'

Custom terminator — Custom terminator

numeric array | integer from 0 to 255, inclusive

Custom terminator that indicates the end of your data block. The simulation considers any data that occurs after the terminator as a new data block. The terminator data is not sent to the output port.

The numeric array specified in this parameter is the uint8 integer representation of the corresponding ASCII characters. The exact form of this parameter depends on the type of the ASCII character.

Type of ASCII Character	Example ASCII Character	MATLAB Command	Parameter Value
Special character	"#"	uint8('#')	[35]
Numeric	"81"	uint8('81')	[56 49]
Alphabet	"End"	uint8('End')	[69 110 100]

You can also specify this parameter using the hexadecimal representation of the ASCII characters.

Programmatic Use
Block Parameter: `CustomTerminator`
Type: character vector, string
Values: integer array

`Data size` — Number of values read

`[1 1]` (default) | numeric array

Output data size, or the number of values that should be read at each simulation time step. This parameter is specified as a multidimensional numeric array. The data size does not include the header or terminator values.

Programmatic Use
Block Parameter: `DataSize`
Type: character vector, string
Values: integer array
Default: `'[1 1]'`

`Enable blocking mode` — Simulation waits while receiving data

`on` (default) | `off`

This parameter has the simulation wait while the block receives data. When new data becomes available, the simulation continues from the next time step. Unselect the check box if you do not want the read operation to cause the simulation to wait.

If you enable blocking mode, the simulation waits for the requested data to become available. The model waits for up to the amount of time specified by the **Timeout** parameter in the Serial Configuration block. If new data does not become available during a simulation, you can return an error by selecting the `Error` option for the **Action when data is not available** parameter.

If you do not enable blocking mode, the simulation runs continuously and the block has two output ports, **Status** and **Data**. The **Data** port contains the requested set of data at each time step. The **Status** port contains 0 or 1 based on whether it received new data at the given time step.

Programmatic Use
Block Parameter: `EnableBlockingMode`
Type: character vector, string
Values: `'on'` | `'off'`
Default: `'on'`

`Action when data is not available` — Action to take when data not available

`Output last received value` (default) | `Output custom value` | `Error`

Action the block should take when data is not available. Available options are:

- `Output last received value` — Block returns the value it received at the preceding time step when it does not receive data at current time step.

- `Output custom value` — Block returns any user-defined value when it does not receive current data. Define the custom value in the **Custom value** field.
- `Error` — Block returns an error when it does not receive current data. You must select **Enable blocking mode** to use this option.

Programmatic Use
Block Parameter: `ActionDataUnavailable`
Type: character vector, string
Values: `'Output last received value'` | `'Output custom value'` | `'Error'`
Default: `'Output last received value'`

`Custom value` — Custom output value when data unavailable

0 (default) | numeric

Custom value for the block to output when it does not receive new data. The custom value can be a scalar or value equal to the size of data that it receives (specified by **Data size** parameter). You must select `Output custom value` as the **Action when data is unavailable** to set this parameter.

Programmatic Use
Block Parameter: `CustomValue`
Type: character vector, string
Values: numeric
Default: `'0'`

`Block sample time` — Sampling time

0.01 (default) | positive numeric

Sampling time of the block during the simulation. This is the rate at which the block is executed during simulation.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector, string
Values: positive numeric
Default: `'0.01'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block generates platform-specific code for the host machine's platform only (Windows, macOS, Linux).

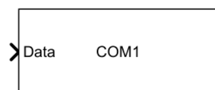
See Also

Serial Configuration | Serial Send

Introduced in R2008a

Serial Send

Send binary data over serial port
Library: Instrument Control Toolbox



Description

The Serial Send block configures and opens an interface to the specified serial port. The configuration and initialization occur once at the start of the model's execution. The block sends data from the model to the serial port during the model's run time. You can use multiple Serial Send blocks at a time to send data to a specific serial port.

Note You must configure your serial port parameters using the Serial Configuration block before you specify the Serial Send block parameters.

The Serial Send block has one input port that accepts both 1-D vector and matrix data. This block has no output ports. The block inherits the data type from the signal at the input port. Valid data types are `single`, `double`, `int8`, `uint8`, `int16`, `uin16`, `int32`, and `uint32`.

Other Supported Features

- The Serial Send block supports the use of Simulink Accelerator mode, but not Rapid Accelerator. This feature speeds up the execution of Simulink models.
- The Serial Send block supports the use of model referencing. This feature lets your model include other Simulink models as modular components.
- The Serial Send block supports C/C++ code generation. This feature allows you to generate C and C++ code using Simulink Coder.

For more information on these features, see the “Simulink” documentation.

Ports

Input

`Data` — Data values to send

vector | matrix | array

Data values to send from the block over your serial port, specified as a vector, matrix, or array. Set the parameters for this block before you send data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Parameters

`Port` — Serial communication port
available communication ports

Serial ports on your machine that you want to send data to. Select a port from the available ports and then configure the port using the Serial Configuration block. If you have not configured a port, the block returns an error when you run your model.

Note Each Serial Send block must have a configured Serial Configuration block. If you use multiple serial ports in your simulation, you must configure each port using a separate Serial Configuration block.

Programmatic Use
Block Parameter: `Port`
Type: character vector, string

`Header` — Header
numeric array | integer from 0 to 255, inclusive

Header that indicates the beginning of your data block. The Serial Send block adds the header in front of the data before sending it over the serial port. By default, no header is specified.

The numeric array specified in this parameter is the `uint8` integer representation of the corresponding ASCII characters. The exact form of this parameter depends on the type of the ASCII character.

Type of ASCII Character	Example ASCII Character	MATLAB Command	Parameter Value
Special character	`"#"`	`uint8('#')`	`[35]`
Numeric	`"81"`	`uint8('81')`	`[56 49]`
Alphabet	`"Start"`	`uint8('Start')`	`[83 116 97 114 116]`

You can also specify this parameter using the hexadecimal representation of the ASCII characters.

Programmatic Use
Block Parameter: `Header`
Type: character vector, string
Values: integer array

`Terminator` — Terminator
`<none>` (default) | `CR ('\r')` | `LF ('\n')` | `CR/LF ('\r\n')` | `NULL ('\0')` | `Custom Terminator`

Terminator that indicates the end of your data block. The Serial Send block appends the terminator to the data before sending it over the serial port.

If you select `Custom Terminator`, you can specify your own terminator value.

Programmatic Use
Block Parameter: `Terminator`
Type: character vector, string
Values: `'<none>'` | `'CR ('\r')'` | `'LF ('\n')'` | `'CR/LF ('\r\n')'` | `'NULL ('\0')'` | `'Custom Terminator'`
Default: `'<none>'`

`Custom terminator` — Custom terminator

numeric array | integer from 0 to 255, inclusive

Custom terminator that indicates the end of your data block. The Serial Send block appends the terminator to the data before sending it over the serial port.

The numeric array specified in this parameter is the `uint8` integer representation of the corresponding ASCII characters. The exact form of this parameter depends on the type of the ASCII character.

Type of ASCII Character	Example ASCII Character	MATLAB Command	Parameter Value
Special character	"#"	uint8('#')	[35]
Numeric	"81"	uint8('81')	[56 49]
Alphabet	"End"	uint8('End')	[69 110 100]

You can also specify this parameter using the hexadecimal representation of the ASCII characters.

Programmatic Use
Block Parameter: `CustomTerminator`
Type: character vector, string
Values: integer array

`Enable blocking mode` — Simulation waits while sending data

on (default) | `off`

This parameter has the simulation wait while the block sends data. Unselect the check box if you do not want the write operation to cause the simulation to wait.

If you enable blocking mode, the simulation waits for the data to be sent. If you do not enable blocking mode, the simulation runs continuously.

Programmatic Use
Block Parameter: `EnableBlockingMode`
Type: character vector, string
Values: `'on'` | `'off'`
Default: `'on'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block generates platform-specific code for the host machine's platform only (Windows, macOS, Linux).

See Also

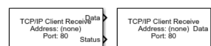
Serial Configuration | Serial Receive

Introduced in R2008a

TCP/IP Receive

Receive data over TCP/IP network from specified remote machine

Library: Instrument Control Toolbox



Description

The TCP/IP Receive block configures and opens an interface to the specified remote address using the TCP/IP protocol. The configuration and initialization occur once at the start of the model's execution. The block acquires data either in blocking mode or nonblocking mode during the model's run time. Use the TCP/IP Receive block to read streaming data over a TCP/IP network. This block works only as a TCP/IP client and cannot be used as a TCP/IP server.

This block has no input ports. It has one or two output ports based on whether you select blocking or nonblocking mode. If you select blocking mode, the block has one output port, **Data**, corresponding to the data it receives. If you do not select blocking mode, the block has two output ports, **Data** and **Status**.

This block uses a First In, First Out (FIFO) buffer to receive data. At each time step, the **Data** port returns the requested values from the buffer. In nonblocking mode, the **Status** port indicates if the block has received new data. If the **Status** port displays `1`, new data is available and if the **Status** port displays `0`, no new data is available.

Other Supported Features

- The TCP/IP Receive block supports the use of Simulink Accelerator mode and Rapid Accelerator. This feature speeds up the execution of Simulink models.
- The TCP/IP Receive block supports the use of model referencing. This feature lets your model include other Simulink models as modular components.
- The TCP/IP Receive block supports C/C++ code generation. This feature allows you to generate C and C++ code using Simulink Coder.

For more information on these features, see the “Simulink” documentation.

Ports

Output

`Data` — Data received
vector | matrix | array

Data received by the block from the remote address, returned as a vector, matrix, or array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`Status` — New data available
`0` (default) | `1`

New data available status, returned as a logical `1` (`true`) or `0` (`false`). If this port returns `1`, new data is available to be read.

Dependencies

To enable this port, unselect the **Enable blocking mode** parameter.

Data Types: `Boolean`

Parameters

`Remote address` — Remote host name or address
remote host name or IP address

IP address or name of the TCP/IP server that you want to receive data from.

Programmatic Use
Block Parameter: `Host`
Type: character vector, string

`Port` — Remote host port
`80` (default) | remote host port

Remote port on the TCP/IP server that you want to connect to, specified as a number from 1 to 65535.

Programmatic Use
Block Parameter: `Port`
Type: character vector, string
Values: 1 to 65,535
Default: `'80'`

`Verify address and port connectivity` — Remote machine connection verification
button

Click this button to check if a connection to the specified remote address and port is valid.

`Data size` — Number of values read
`[1,1]` (default) | numeric array

Output data size, or the number of values to be read at each simulation time step. This parameter is specified as a scalar or vector. The data does not include the terminator values.

Programmatic Use
Block Parameter: `DataSize`
Type: character vector, string
Values: vector
Default: `'[ 1, 1 ]'`

`Source Data type` — Output data type
`uint8` (default) | `single` | `double` | `int8` | `int16` | `uint16` | `int32` | `uint32` | `int64` | `uint64` | `ASCII`

Data type that the block receives from the remote address, specified as a MATLAB data type.

This data type must match the data type of the data at the remote address. You cannot use this parameter to change the data type of the data at the remote address.

Programmatic Use
Block Parameter: `DataType`
Type: character vector, string
Values: `'single'` | `'double'` | `'int8'` | `'uint8'` | `'int16'` | `'uint16'` | `'int32'` | `'uint32'` | `'int64'` | `'uint64'` | `'ASCII'`
Default: `'uint8'`

`ASCII format string` — Format of string data

`%f` (default) | numeric conversion specifiers

This parameter defines the format of the received string data. You can use the following conversion specifiers or a combination of them.

Note If you generate C/C++ code using Simulink Coder, you can only use a single conversion specifier.

Numeric Field Type	Conversion Specifier	Details
Integer, signed	`%d`	Base 10
	`%i`	The values in the data determine the base: • The default is base 10. • If the initial digits are `0x` or `0X`, then the values are hexadecimal (base 16). • If the initial digit is `0`, then values are octal (base 8).
	`%ld` or `%li`	64-bit values, base 10, 8, or 16
Integer, unsigned	`%u`	Base 10
	`%o`	Base 8 (octal)
	`%x`	Base 16 (hexadecimal)
	`%lu`, `%lo`, `%lx`	64-bit values, base 10, 8, or 16
Floating-point number	`%f`	Floating-point fields can contain any of the following (not case sensitive): `Inf`, `-Inf`, `NaN`, or `-NaN`.
	`%e`	
	`%g`	

Programmatic Use
Block Parameter: `ASCIIFormatting`
Type: character vector, string
Values: `'%d'` | `'%i'` | `'%ld'` | `'%li'` | `'%u'` | `%o'` | `'%x'` | `'%lu'` | `'%lo'` | `'%lx'` | `'%f'` | `'%e'` | `'%g'`
Default: `'%f'`

Dependencies

To enable this parameter, set **Source Data type** to `ASCII`.

`Terminator` — Terminator

`LF` (default) | `CR` | `CR/LF` | `LF/CR` | `Custom terminator`

If this parameter is selected, you can specify the terminator that indicates the end of your data block. The simulation considers any data that occurs before the terminator as a new data block. The terminator data is not sent to the output port. This terminator must match the terminator in the data you are reading from your remote machine.

If you select `Custom Terminator`, you can specify your own terminator value.

Programmatic Use
Block Parameter: `Terminator`
Type: character vector, string
Values: `'CR'` | `'LF'` | `'CR/LF'` | `'LF/CR'` | `'Custom terminator'`
Default: `'LF'`

Dependencies

To enable this parameter, set **Source Data type** to `ASCII`.

`Custom Terminator` — Custom terminator

numeric array | integer from 0 to 255

Custom terminator that indicates the end of your data block. The simulation considers any data that occurs before the terminator as a new data block. The terminator data is not sent to the output port.

The numeric array specified in this parameter is the `uint8` integer representation of the corresponding ASCII characters. The exact form of this parameter depends on the type of the ASCII character.

Type of ASCII Character	Example ASCII Character	MATLAB Command	Parameter Value
Control character (escape sequence)	`"LF"` (`"\n"`)	`uint8(sprintf('\n'))`	`[10]`
Special character	`"#"`	`uint8('#')`	`[35]`
Numeric	`"81"`	`uint8('81')`	`[56 49]`
Alphabet	`"End"`	`uint8('End')`	`[69 110 100]`

You can also specify this parameter using the hexadecimal representation of the ASCII characters.

Programmatic Use
Block Parameter: `CustomTerminator`
Type: character vector, string
Values: integer array
Default: `'10'`

Dependencies

To enable this parameter, set **Terminator** to `Custom terminator`.

`Byte order` — Sequential order of bytes

`big-endian` (default) | `little-endian`

Sequential order in which bytes are arranged into larger numerical values. If the byte order is `little-endian`, then the first byte is organized in the first memory address in the received TCP/IP packet. If the byte order is `big-endian`, then the last byte is organized in the first memory address in the received TCP/IP packet.

Configure the byte order to match the appropriate value for your remote machine before receiving data. Refer to your instrument documentation for information about the order in which it stores bytes.

Programmatic Use
Block Parameter: `ByteOrder`
Type: character vector, string
Values: `'little-endian'` | `'big-endian'`
Default: `'big-endian'`

`Enable blocking mode` — Simulation waits while receiving data

`on` (default) | `off`

This parameter has the simulation wait while the block receives data. When new data becomes available, the simulation continues from the next time step. Unselect the check box if you do not want the read operation to cause the simulation to wait.

If you enable blocking mode, the simulation waits for the requested data to become available. At each time step, the model waits for up to the amount of time specified by the **Timeout** parameter. If data is not received within this time, the block outputs a value of `0`.

If you do not enable blocking mode, the simulation runs continuously and the block has two output ports, **Status** and **Data**. The **Data** port contains the requested set of data at each time step. The **Status** port contains 0 or 1 based on whether it received new data at the given time step.

For more information, see “Enable Blocking Mode in Receive and Send Blocks” on page 23-25.

Programmatic Use
Block Parameter: `EnableBlockingMode`
Type: character vector, string
Values: `'on'` | `'off'`
Default: `'on'`

`Timeout` — Allowed time to complete operations

10 (default) | positive scalar

Amount of time in seconds that the model waits for data during each simulation time step. This value is relative to real-world or "wall clock" time.

Programmatic Use
Block Parameter: `Timeout`
Type: character vector, string
Values: positive scalar
Default: `'10'`

Dependencies

To enable this parameter, set **Enable blocking mode** to `on`.

`Block sample time` — Sampling time

0.01 (default) | positive numeric

Sampling time of the block during the simulation. This is the rate at which the block is executed during simulation. For more information, see “Timing in Hardware Interface Models” on page 23-30.

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector, string
Values: positive numeric
Default: `'0.01'`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

This block generates platform-specific code for the host machine's platform only (Windows, macOS, Linux). Set the **Device vendor** and **Device type** in **Model Settings > Hardware Implementation**. You must also select **Support long long** under **Device details**.

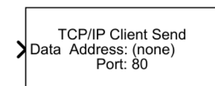
You can only use a single conversion specifier for the **ASCII format string** parameter.

See Also
TCP/IP Send

Introduced in R2007b

TCP/IP Send

Send data over TCP/IP network to specified remote machine
Library: Instrument Control Toolbox



Description

The TCP/IP Send block configures and opens an interface to the specified remote address using the TCP/IP protocol. The configuration and initialization occur once at the start of the model's execution. The block sends data either in blocking mode or nonblocking mode during the model's run time. Use the TCP/IP Send block to write streaming data over a TCP/IP network. This block works only as a TCP/IP client and cannot be used as a TCP/IP server.

The TCP/IP Send block has one input port that accepts both 1-D vector and matrix data. This block has no output ports. The block inherits the data type from the signal at the input port. Valid data types are `single`, `double`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, and `uint64`.

Other Supported Features

- The TCP/IP Send block supports the use of Simulink Accelerator mode and Rapid Accelerator. This feature speeds up the execution of Simulink models.
- The TCP/IP Send block supports the use of model referencing. This feature lets your model include other Simulink models as modular components.
- The TCP/IP Send block supports C/C++ code generation. This feature allows you to generate C and C++ code using Simulink Coder.

For more information on these features, see the “Simulink” documentation.

Ports

Input

`Data` — Data values to send
vector | matrix | array

Data values to send from the block to your remote host, specified as a vector, matrix, or array. Set the parameters for this block before you send data.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Parameters

`Remote address` — Remote host name or address
remote host name or IP address

IP address or name of the TCP/IP server that you want to send data to.

Programmatic Use
Block Parameter: `Host`
Type: character vector, string

`Port` — Remote host port

`80` (default) | remote host port

Remote port on the TCP/IP server that you want to connect to, specified as a number from 1 to 65535.

Programmatic Use
Block Parameter: `Port`
Type: character vector, string
Values: 1 to 65,535
Default: `'80'`

`Verify address and port connectivity` — Remote machine connection verification

button

Click this button to check if a connection to the specified remote address and port is valid.

`Byte order` — Sequential order of bytes

`big-endian` (default) | `little-endian`

Sequential order in which bytes are arranged into larger numerical values. If the byte order is `little-endian`, then the remote machine stores the first byte in the first memory address. If the byte order is `big-endian`, then the remote machine stores the last byte in the first memory address.

Configure the byte order to match the appropriate value for your remote machine before sending data. Refer to your instrument documentation for information about the order in which it stores bytes.

Programmatic Use
Block Parameter: `ByteOrder`
Type: character vector, string
Values: `'little-endian'` | `'big-endian'`
Default: `'big-endian'`

`Enable blocking mode` — Simulation waits while sending data

`on` (default) | `off`

This parameter has the simulation wait while the block sends data. Unselect the check box if you do not want the write operation to cause the simulation to wait.

If you enable blocking mode, the simulation waits for the data to be sent. If you do not enable blocking mode, the simulation runs continuously.

For more information, see "Enable Blocking Mode in Receive and Send Blocks" on page 23-25.

Programmatic Use
Block Parameter: `EnableBlockingMode`
Type: character vector, string
Values: `'on'` | `'off'`
Default: `'on'`

`Timeout` — Allowed time to complete operations

10 (default) | positive scalar

Amount of time in seconds that the model waits for data to be sent during each simulation time step. This value is relative to real-world or "wall clock" time.

Programmatic Use
Block Parameter: `Timeout`
Type: character vector, string
Values: positive scalar
Default: `'10'`

Dependencies

To enable this parameter, set **Enable blocking mode** to `on`.

`Transfer Delay` — Enables delayed acknowledgement

on (default) | `off`

If you enable this parameter, the block collects small segments of outstanding data and sends them in a single packet when acknowledgement (ACK) arrives from the server. Unselect this check box if you want to immediately send data to the network.

If a network is slow, you can improve its performance by enabling the transfer delay. However, on a fast network acknowledgments arrive quickly and the difference between enabling or disabling the transfer delay is negligible.

Programmatic Use
Block Parameter: `TransferDelay`
Type: character vector, string
Values: `'on'` | `'off'`
Default: `'on'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block generates platform-specific code for the host machine's platform only (Windows, macOS, Linux). Set the **Device vendor** and **Device type** in **Model Settings > Hardware Implementation**. You must also select **Support long long** under **Device details**.

See Also

TCP/IP Receive

Introduced in R2007b

To Instrument

Send simulation data to instrument

Library

Instrument Control Toolbox

Description

The To Instrument block configures and opens an interface to an instrument, initializes the instrument, and sends data to the instrument. The configuration and initialization happen at the start of the model execution. The block sends data to the instrument during model run time.

The block has no output ports. The block has one input port corresponding to the data sent to the instrument. This data type must be double precision.

Note The To Instrument block can be used with these interfaces: VISA, GPIB, Serial, TCP/IP, and UDP. It is not supported on these interfaces: SPI, I2C, and Bluetooth.

Other Supported Features

- This block supports the use of Simulink Accelerator mode, but not Rapid Accelerator or code generation.
- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the "Simulink" documentation.

Parameters

Block sample time

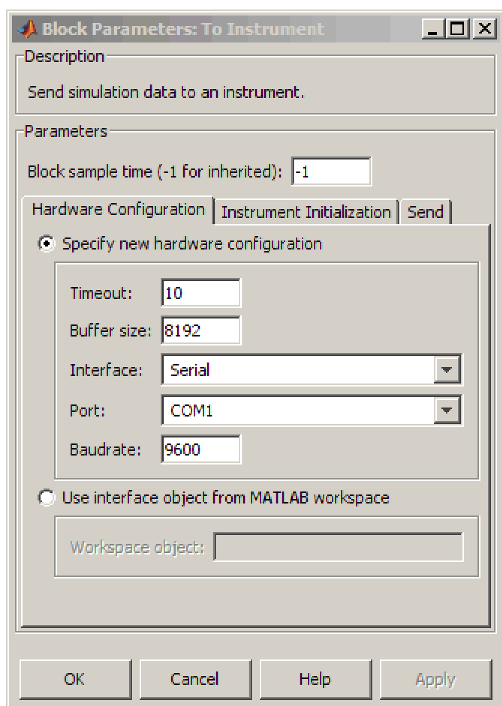
The Block sample time parameter is the only setting outside of the dialog box tabs. The default value of `-1` sets the block to inherit timing. A positive value is used as the sample period.

Hardware Configuration Tab

The **Hardware Configuration** tab is where you define the settings for communications with your instrument. You have two choices about establishing an interface:

- Specify a new hardware configuration.
- Use an interface object from the MATLAB workspace.

The following figure shows the **Hardware Configuration** tab set to specify a new hardware configuration using a serial port interface.



Because some parameters apply to multiple interface types, they appear here in alphabetical order.

Baudrate

The rate at which bits are transmitted for the serial or VISA serial interface.

Board index

The index of the board used for GPIB, VISA GPIB, VISA TCPIP, or VISA USB interface to the instrument. See BoardIndex property for more information.

Board vendor

The vendor of the GPIB board used for the interface to the instrument. Your choices are Keysight (formerly Agilent), ICS Electronics, Measurement Computing (MCC), and National Instruments.

Chassis index

The index number of the VXI chassis. Used for VISA VXI and VISA VXI-GPIB interface types.

Buffer size

The total number of bytes that can be stored in the software output buffer during a read operation.

Interface

Select the type of hardware interface to the instrument. Your options are those interfaces supported by the Instrument Control Toolbox software. The previous figure shows a configuration for a serial port interface.

Logical address

The logical address of the VXI instrument. Used for VISA VXI and VISA VXI-GPIB interface types.

Manufacturer ID

The manufacturer ID of the VISA USB instrument defined as a character vector. See ManufacturerID property for more information.

Model code

The model code of the VISA USB instrument defined as a character vector. See ModelCode property for more information.

Port

The port for the serial interface: COM1, COM2, etc.

Primary address

The primary address of the instrument on the GPIB.

Remote host

The host name or IP address of the instrument. Used for UDP, TCPIP, or VISA TCPIP interface types.

Remote port

The port on the instrument or remote host used for communication. Used for UDP, TCPIP, or VISA TCPIP interface types.

Secondary address

The secondary address of the instrument on the GPIB.

Serial number

The serial number of the VISA USB instrument defined as a character vector. See SerialNumber property for more information.

Timeout

Time in seconds, allowed to complete the query operation.

VISA vendor

The vendor of the VISA instrument used for any of the VISA interface types. Your choices are Keysight (formerly Agilent), National Instruments, and Tektronix.

Use interface object from MATLAB workspace

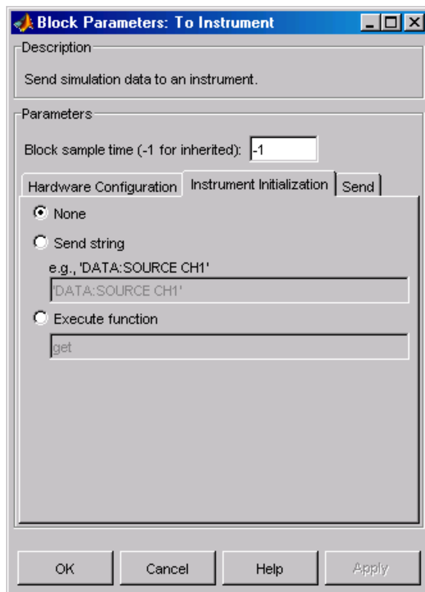
Select this option to use an interface object from the MATLAB workspace.

Workspace object

Enter the object name that you want to use from the MATLAB workspace.

Instrument Initialization Tab

The **Instrument Initialization** tab is where you define what happens when you first open your connection to the instrument.



None

The default initialization option is none.

Send string

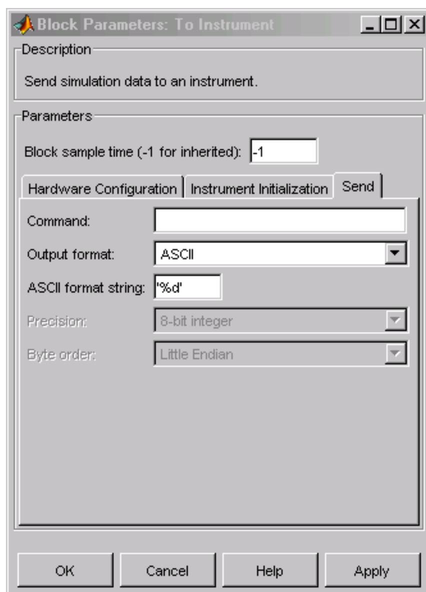
A string sent to the instrument as an instrument command to initialize the instrument or set it up in a known state.

Execute function

Any function that has as its only argument the interface object representing the instrument. You can write this function to include several instrument commands and initialization data.

Send Tab

The **Send** tab is where you define the optional command sent to the instrument and the format of the sent data.



Command

This is the command that is sent to the instrument with the Simulink data. *This command is optional*—if you leave this field blank, the Simulink data is sent to the instrument without any prefix or additional formatting.

Output format

Your options are `ASCII`, `Binary`, or `Binblock` (binary block — the `binblock` format is described in the `binblockwrite` function reference page).

ASCII format string

Available only when the format is ASCII, this defines the format string for the data. For a list of formats, see the `fprintf` function.

Precision

Used for binary or binblock format. Your options are:

- `8-bit integer` (default)
- `16-bit integer`
- `32-bit integer`
- `8-bit unsigned integer`
- `16-bit unsigned integer`
- `32-bit unsigned integer`
- `32-bit float`
- `64-bit float`

Byte order

When using binary or binblock format with more than 8 bits, you can specify the instrument's byte order for the data. Your options are `Big Endian` or `Little Endian`.

Tip Hardware information shown in the dialog box is determined and cached when you first open the dialog box. To refresh the display with new values, restart MATLAB.

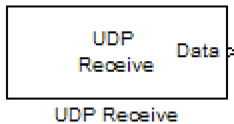
See Also

Query Instrument, TCP/IP Receive, TCP/IP Send, UDP Receive, UDP Send

Introduced before R2006a

UDP Receive

Receive data over UDP network from specified remote machine



UDP Receive

Library

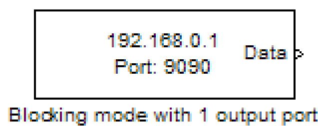
Instrument Control Toolbox

Description

The UDP Receive block configures and opens an interface to a specified remote address using the UDP protocol. The configuration and initialization occur once at the start of the model's execution. During the model's run time, the block acquires data either in blocking or nonblocking mode.

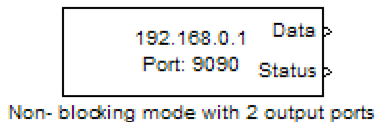
Note You need a license for both the Instrument Control Toolbox and Simulink software to use this block.

This block has no input ports. It has either one or two output ports based on your selection of blocking or nonblocking mode. If you select blocking mode, the block will have one output port corresponding to the data it receives.



Blocking mode with 1 output port

If you do not select blocking mode, the block will have two output ports, the **Data** port and the **Status** port.



Non- blocking mode with 2 output ports

A First In First Out (FIFO) buffer receives the data. At every time step, the **Data** port outputs the requested values from the buffer. In a nonblocking mode, the **Status** port indicates if the block has received new data.

Other Supported Features

- This block supports the use of Simulink Accelerator mode, but not Rapid Accelerator or code generation.
- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the “Simulink” documentation.

Parameters

Note You can enter MATLAB variables in the text edit fields in the UDP Receive Block Parameters dialog box, except for these fields: **Local address**, **Remote address**, **ASCII format string**, and **Terminator**.

Local address

Specify the IP address, name, or the Web server address of the local host. This is the same as the UDP interface `localhost` property. This field is empty by default.

Local port

Specify the port to bind on the local machine. The default value is `-1`, which automatically binds to an available port.

Enable local port sharing

Use to enable port sharing. UDP ports can be shared by other applications to allow for multiple applications to listen to the UDP datagrams on that port. You can bind a UDP object to a specific `LocalPort` number, and in another application bind a UDP socket to that same local port number so both can receive UDP broadcast data. Enabling this option allows other UDP sockets to bind to the UDP object’s `LocalPort`. It is off by default.

Remote address

Specify the IP address, name, or the Web server address of the machine from which you need to receive data. This field is set to `localhost` by default.

Remote Port

Specify the remote port on the host you need to connect to. The default port value is `9090`. Valid port values are `1` to `65535`.

Verify address and port connectivity

Click this button to:

- Check if the specified remote address is correct.
- Establish connection with the specified remote address and port.

Output latest data

Enable to receive the latest data from the UDP Receive block. Data is normally received in a FIFO manner. When enabled, the latest available packets are received, instead of using the FIFO method. Note that this option is disabled when data type is ASCII. This option is off by default.

Data size

Specify the output data size, or the number of values that should be read at each simulation time step. This parameter is specified as a row vector of nonnegative integers, where each element represents the length of the corresponding dimension. For example, if you specify the data size as `[m n]`, `m` represents the number of rows and `n` represents the number of columns. The data size does not include the terminator value. The default value is `[1 1]`, which represents a single scalar value.

Source Data type

Specify the input data type to receive from the block. You can select from the following values:

- `single`
- `double`
- `int8`
- `uint8` (default)
- `int16`
- `uint16`
- `int32`
- `uint32`
- `ASCII`

ASCII format string

This option is only available when you select ASCII as your data type. It defines the format string for the data. For a list of formats, see the `fscanf` function.

Terminator

This option is only available when you select ASCII as your data type. It can be used to set a terminator for the data read. For more information about setting the property, see Terminator.

Byte order

When using binary or binblock format with more than 8 bits, you can specify the instrument's byte order for the data. Your options are `BigEndian` or `LittleEndian`.

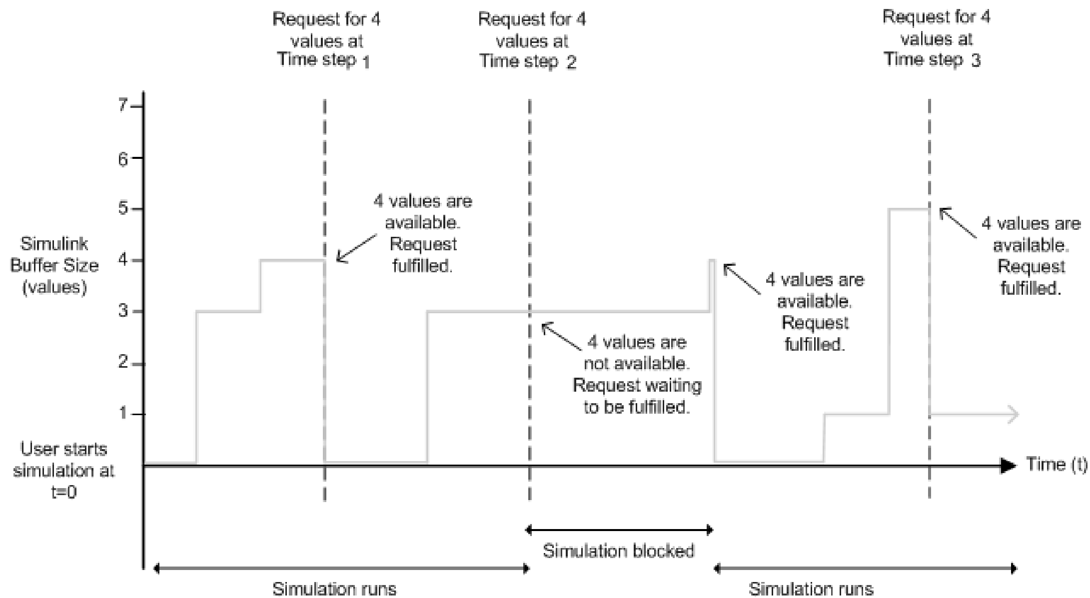
Enable blocking mode

Specify if you want to block the simulation while receiving data. This option is selected by default. Clear this check box if you do not want the read operation to block the simulation.

If you enable blocking mode, the model will block the simulation while it is waiting for the requested data to be available. If you do not enable blocking mode, the simulation runs continuously. The block has two output ports, **Status** and **Data**. The **Data** port contains the requested set of data at each time step. The **Status** port contains 0 or 1 based on whether it received new data at the given time step. The following diagrams show the difference between receiving data using blocking mode and nonblocking mode.

In this example, you start the simulation at time (t=0) and specify the amount of data to receive as 4 (set in the **Data size** field of the UDP Receive Block Parameters dialog box). After the simulation starts, the data is acquired asynchronously in a FIFO buffer.

Blocking Mode



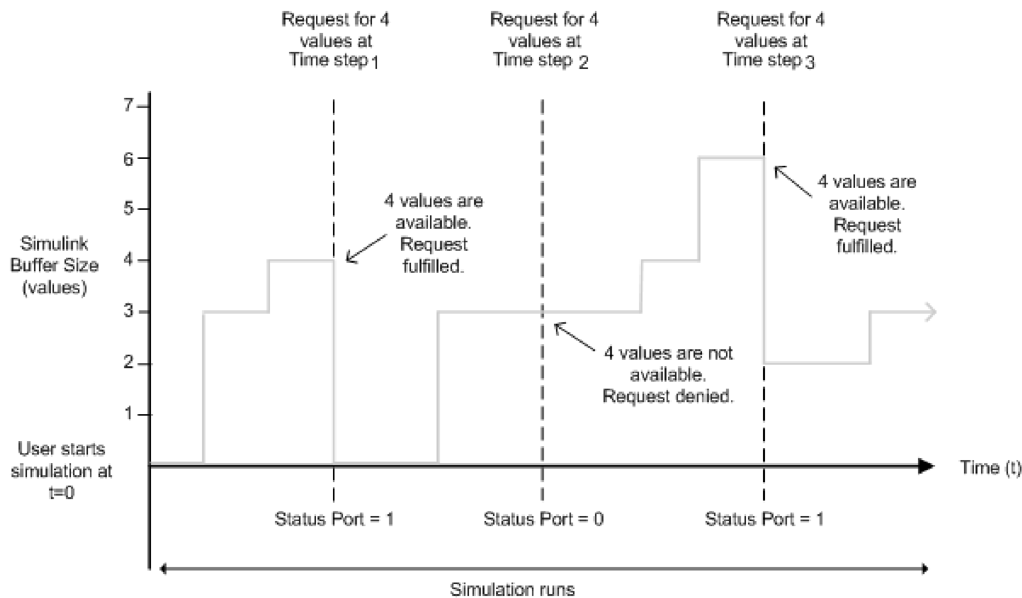
The blocking mode simulation occurs like this:

- At time step 1: The Simulink software requests data and the buffer has four values available, the block fulfills the request without interrupting the simulation. The block resets the buffer value to 0.
- At time step 2: The Simulink software requests data again, and the buffer has only three values, therefore it blocks the simulation until it receives the fourth value. When the block receives the fourth value, it fulfills the request and resumes the simulation. The block resets the buffer value to 0.
- At time step 3: When the Simulink software requests data, the block has five values and it returns the first four that it received and resets the buffer to 1.

If the requested data is not received within the amount of time specified in the **Timeout** field (of the UDP Receive Block Parameters dialog box), a Simulink software error occurs and the simulation is stopped.

Note In blocking mode, if you have more than one UDP model on your computer, ensure that the Receive block is receiving data. If it is not, then your model might error out. You can avoid this by either changing the block to Nonblocking mode or by resetting the block's priority.

Nonblocking Mode



Here the simulation is not blocked and runs continuously.

- At time step 1: The Simulink software requests data and the buffer has four values available, the block fulfills the request and changes the `Status` port value to `1`, indicating that new data is available. The `Data` port at this point contains the newly received values. The block resets the buffer value to 0.
- At time step 2: The Simulink software requests data again, and the buffer has only three values, and the block cannot return it as data size is specified as 4. Therefore the block sets the `Status` port value to `0`, indicating that there is no new data. The `Data` port contains the previously received value, and the buffer is at three (the number of values it has received since the last request was fulfilled).
- At time step 3: When the Simulink software requests data here, the buffer now has five values and it returns the first four in the order it received and changes the `Status` port value to `1`.

Timeout

Specify the amount of time that the model will wait for the data during each simulation time step. The default value is `inf` (seconds). This field is unavailable if you have not enabled blocking mode.

Block sample time

Specify the sampling time of the block during simulation. The default value is `0.01` (seconds).

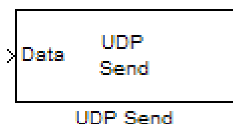
See Also

Query Instrument, Serial Configuration, Serial Receive, Serial Send, TCP/IP Receive, TCP/IP Send, To Instrument, UDP Send

Introduced in R2007b

UDP Send

Send data over UDP network to specified remote machine



Library

Instrument Control Toolbox

Description

The UDP Send block sends data from your model to the specified remote machine using the UDP protocol.

Note You need a license for both the Instrument Control Toolbox and Simulink software to use this block.

The UDP Send block has one input port and it accepts both 1-D vector and matrix data. This block has no output ports. The block inherits the data type from the signal at the input port.

Other Supported Features

- This block supports the use of Simulink Accelerator mode, but not Rapid Accelerator or code generation.
- The block supports the use of model referencing, so that your model can include other Simulink models as modular components.

For more information on these features, see the “Simulink” documentation.

Parameters

Note You can enter MATLAB variables in the text edit fields in the UDP Send Block Parameters dialog box, except for these fields: **Local address** and **Remote address**.

Remote address

Specify the IP address, name, or the Web server address of the machine to which you need to send data. This field is empty by default.

Remote port

Specify the remote port on the host you need to send the data to. The default port value is `9090`. Valid port values are `1` to `65535`.

Local address

Specify the IP address, name, or the Web server address of the local host. This is the same as the UDP interface `localhost` property. This field is empty by default.

Local port

Specify the port to bind on the local machine. The default value is `-1`, which automatically binds to an available port.

Enable local port sharing

Use to enable port sharing. UDP ports can be shared by other applications to allow for multiple applications to listen to the UDP datagrams on that port. You can bind a UDP object to a specific `LocalPort` number, and in another application bind a UDP socket to that same local port number so both can receive UDP broadcast data. Enabling this option allows other UDP sockets to bind to the UDP object's `LocalPort`. It is off by default.

Verify address and port connectivity

Click this button to:

- Check if the specified remote address is correct.
- Establish connection with the specified remote address and port.

UDP packet size

Use to set the `OutputDatagramPacketSize` property. The UDP packet size is controlled by the `OutputDatagramPacketSize` property. You can specify the size, in bytes, between 1 and 65,535, and the default value is 512. You can increase or decrease the packet size if necessary.

Byte order

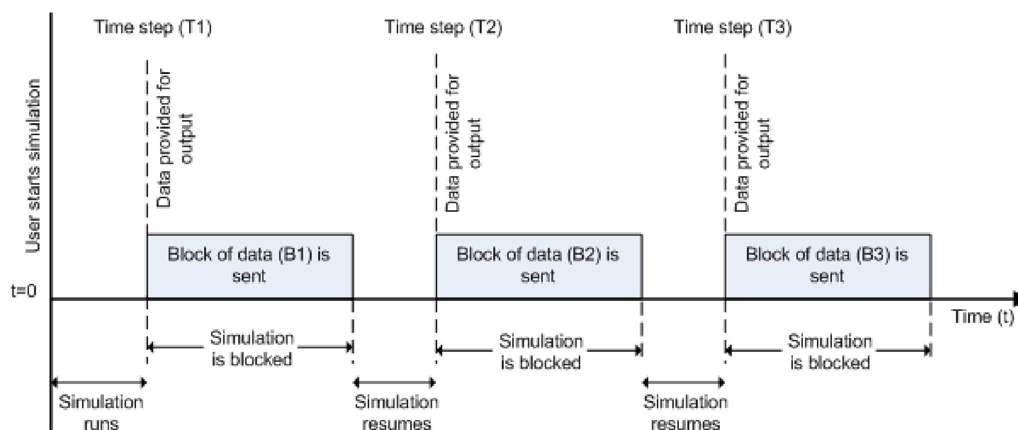
When using binary or binblock format with more than 8 bits, you can specify the instrument's byte order for the data. Your options are `BigEndian` or `LittleEndian`.

Enable blocking mode

Specify if you want to block the simulation while sending data. This option is selected by default. Clear this check box if you do not want the write operation to block the simulation.

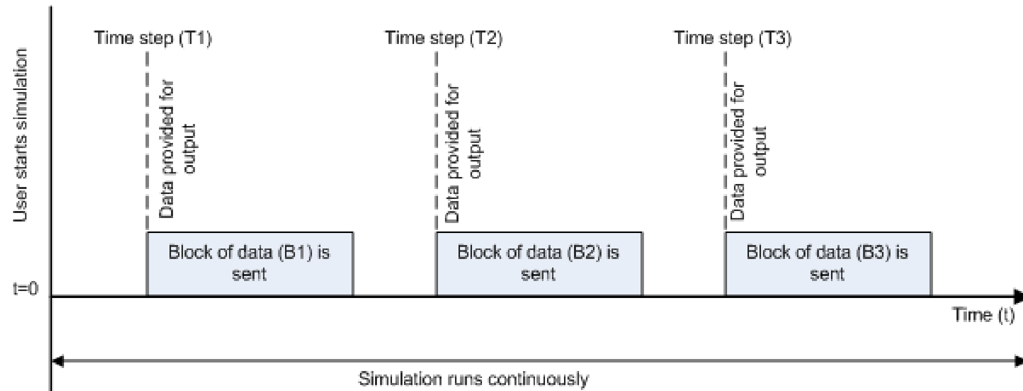
The following diagrams show the difference between sending data using blocking mode and nonblocking mode.

Blocking Mode



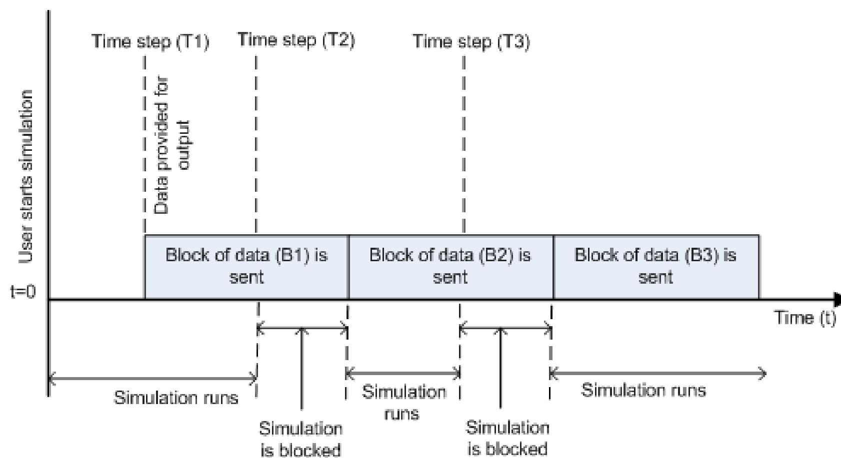
In this example, you start the simulation at time (t=0). At time step (T1), data output is initiated and simulation stops until the block of data (B1) is sent to the specified remote address and port. After the data is sent, simulation resumes until time step (T2), where the block initiates another data output and simulation is blocked until the block of data (B2) is sent to the remote address and port, and the simulation resumes.

Nonblocking Mode – Scenario 1



In this scenario, the data output outpaces the simulation speed. Data output is initiated at the first time step (T1) and the corresponding block of data (B1) is sent to the specified remote address asynchronously. The simulation runs continuously in this mode.

Nonblocking Mode – Scenario 2



In this scenario, the simulation is nonblocking and occurs faster than the data initiation.

- At time step T1: The block of data (B1) is sent to the specified remote address and port asynchronously.

- At time step T2: The simulation is blocked until the block of data (B1) is sent completely. When the (B1) is completely sent, the new block of data (B2) is sent asynchronously, and the simulation resumes.

Note Several factors, including network connectivity and model complexity, can affect the simulation speed. This can cause both nonblocking scenarios to occur within the same simulation.

See Also

Query Instrument, Serial Configuration, Serial Receive, Serial Send, TCP/IP Receive, TCP/IP Send, To Instrument, UDP Receive

Introduced in R2007b

A

Bibliography

[1] Axelson, Jan, *Serial Port Complete*, Lakeview Research, Madison, WI, 1998.

[2] *Courier High Speed Modems User's Manual*, U.S. Robotics, Inc., Skokie, IL, 1994.

[3] TIA/EIA-232-F, *Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*.

[4] *Getting Started with Your AT Serial Hardware and Software for Windows 98/95*, National Instruments, Inc., Austin, TX, 1998.

[5] *HP E1432A User's Guide*, Hewlett-Packard Company, Palo Alto, CA, 1997.

[6] *HP 33120A Function Generator/Arbitrary Waveform Generator User's Guide*, Hewlett-Packard Company, Palo Alto, CA, 1997.

[7] *HP VISA User's Guide*, Hewlett-Packard Company, Palo Alto, CA, 1998.

[8] *NI-488.2M™ User Manual for Windows 95 and Windows NT*, National Instruments, Inc., Austin, TX, 1996.

[9] *NI-VISA™ User Manual*, National Instruments, Inc., Austin, TX, 1998.

[10] IEEE Std 488.2-1992, *IEEE Standard Codes, Formats, Protocols, and Common Commands for Use with IEEE Std 4881.-1987, IEEE Standard Digital Interface for Programmable Instrumentation*, Institute of Electrical and Electronics Engineers, New York, NY, 1992.

[11] *Instrument Communication Handbook*, IOTech, Inc., Cleveland, OH, 1991.

[12] *TDS 200-Series Two Channel Digital Oscilloscope Programmer Manual,* Tektronix, Inc., Wilsonville, OR.

[13] Stevens, W. Richard, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, Boston, MA, 1994.