How to Contact The MathWorks

www.mathworks.com   Web
comp.soft-sys.matlab   Newsgroup
www.mathworks.com/contact_TS.html   Technical Support
suggest@mathworks.com   Product enhancement suggestions
bugs@mathworks.com   Bug reports
doc@mathworks.com   Documentation error reports
service@mathworks.com   Order status, license renewals, passcodes
info@mathworks.com   Sales, pricing, and general information

508-647-7000 (Phone)
508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

Instrument Control Toolbox User’s Guide


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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Patents

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

<table>
<thead>
<tr>
<th>Date</th>
<th>Printing Status</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2000</td>
<td>First printing</td>
<td>New for Version 1.0 (Release 12)</td>
</tr>
<tr>
<td>June 2001</td>
<td>Second printing</td>
<td>Revised for Version 1.1 (Release 12.1)</td>
</tr>
<tr>
<td>July 2002</td>
<td>Online only</td>
<td>Revised for Version 1.2 (Release 13)</td>
</tr>
<tr>
<td>August 2002</td>
<td>Third printing</td>
<td>Revised for Version 1.2</td>
</tr>
<tr>
<td>June 2004</td>
<td>Online only</td>
<td>Revised for Version 2.0 (Release 14)</td>
</tr>
<tr>
<td>October 2004</td>
<td>Fourth printing</td>
<td>Revised for Version 2.1 (Release 14SP1)</td>
</tr>
<tr>
<td>March 2005</td>
<td>Online only</td>
<td>Revised for Version 2.2 (Release 14SP2)</td>
</tr>
<tr>
<td>June 2005</td>
<td>Fifth printing</td>
<td>Minor revision for Version 2.2</td>
</tr>
<tr>
<td>September 2005</td>
<td>Online only</td>
<td>Revised for Version 2.3 (Release 14SP3)</td>
</tr>
<tr>
<td>March 2006</td>
<td>Online only</td>
<td>Revised for Version 2.4 (Release 2006a)</td>
</tr>
<tr>
<td>September 2006</td>
<td>Online only</td>
<td>Revised for Version 2.4.1 (Release 2006b)</td>
</tr>
<tr>
<td>March 2007</td>
<td>Online only</td>
<td>Revised for Version 2.4.2 (Release 2007a)</td>
</tr>
<tr>
<td>September 2007</td>
<td>Sixth printing</td>
<td>Revised for Version 2.5 (Release 2007b)</td>
</tr>
</tbody>
</table>
## Getting Started

### What Is Instrument Control Toolbox? .......................... 1-2
- Getting to Know the Instrument Control Toolbox .......................... 1-2
- Exploring Instrument Control Toolbox ........................................ 1-3
- Learning About Instrument Control Toolbox .................................. 1-3
- Using the Documentation Examples ........................................... 1-4

### Toolbox Components ................................................. 1-5
- Passing Information Between MATLAB and Your Instrument .......... 1-5
- M-File Functions .......................................................... 1-7
- Interface Driver Adaptor ...................................................... 1-8

### Understanding the Toolbox Capabilities .......................... 1-9
- Overview Help ............................................................... 1-9
- Documentation Examples .................................................... 1-9
- Demos .............................................................................. 1-9

### Installation Information .................................................. 1-10
- Installation Requirements ...................................................... 1-10
- Toolbox Installation ........................................................... 1-10
- Hardware and Driver Installation ............................................ 1-11

### Examining Your Hardware Resources .................................. 1-12
- instrhwinfo Function .......................................................... 1-12
- Test & Measurement Tool ...................................................... 1-16
- Viewing the IVI Configuration Store ........................................ 1-18

### Communicating with Your Instrument ............................... 1-21
- Instrument Control Session Examples ....................................... 1-21
- Example: Communicating with a GPIB Instrument ....................... 1-21
- Example: Communicating with a GPIB-VXI Instrument .................. 1-22
- Example: Communicating with a Serial Port Instrument .................. 1-24
Using Interface Objects

Creating an Interface Object ........................................ 3-2
  Object Creation Functions ............................................ 3-2
  Configuring Properties During Object Creation .................. 3-3
  Creating an Array of Instrument Objects .......................... 3-3

Connecting to the Instrument ........................................ 3-5

Configuring and Returning Properties ................................ 3-6
  Base and Interface-Specific Properties .............................. 3-6
  Returning Property Names and Property Values ..................... 3-6
  Configuring Property Values ........................................... 3-9
  Specifying Property Names ............................................ 3-10
  Default Property Values ............................................... 3-10
  Property Inspector ..................................................... 3-11

Writing and Reading Data ........................................... 3-12
  Before Performing Read/Write Operations ........................... 3-12
  Writing Data .......................................................... 3-13
  Reading Data .......................................................... 3-19

Disconnecting and Cleaning Up ..................................... 3-25
  Disconnecting an Instrument Object ................................ 3-25
  Cleaning Up the MATLAB Environment ............................... 3-25

Controlling Instruments Using GPIB

GPIB Overview .......................................................... 4-3
What Is GPIB? ........................................... 4-3
Important GPIB Features ................................ 4-4
GPIB Lines ................................................. 4-5
Status and Event Reporting .............................. 4-10

Creating a GPIB Object .................................... 4-15
Using the gpib Function .................................. 4-15
GPIB Object Display ........................................ 4-16

Configuring the GPIB Address ............................ 4-18

Writing and Reading Data ................................ 4-19
Rules for Completing Write and Read Operations .... 4-19
Example: Writing and Reading Text Data ............... 4-20
Example: Reading Binary Data ............................ 4-22
Example: Parsing Input Data Using scanstr .......... 4-24
Example: Understanding EOI and EOS ................. 4-25

Events and Callbacks .................................... 4-28
Example: Introduction to Events and Callbacks ....... 4-28
Event Types and Callback Properties .................... 4-29
Storing Event Information ................................ 4-30
Creating and Executing Callback Functions ............ 4-32
Enabling Callback Functions After They Error ....... 4-33
Example: Using Events and Callbacks to Read Binary Data ................. 4-33

Triggers .................................................. 4-36
Using the trigger Function ................................ 4-36
Example: Executing a Trigger ............................ 4-36

Serial Polls ................................................ 4-39
Using the spoll Function .................................. 4-39
Example: Executing a Serial Poll ........................ 4-39
# Controlling Instruments Using VISA

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VISA Overview</td>
<td>5-3</td>
</tr>
<tr>
<td>What Is VISA?</td>
<td>5-3</td>
</tr>
<tr>
<td>Interfaces Used with VISA</td>
<td>5-3</td>
</tr>
</tbody>
</table>

## Working with the GPIB Interface
<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding VISA-GPIB</td>
<td>5-5</td>
</tr>
<tr>
<td>Creating a VISA-GPIB Object</td>
<td>5-5</td>
</tr>
<tr>
<td>The VISA-GPIB Address</td>
<td>5-7</td>
</tr>
</tbody>
</table>

## Working with the VXI Interface
<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding VISA-VXI</td>
<td>5-9</td>
</tr>
<tr>
<td>Creating a VISA-VXI Object</td>
<td>5-10</td>
</tr>
<tr>
<td>The VISA-VXI Address</td>
<td>5-12</td>
</tr>
<tr>
<td>Register-Based Communication</td>
<td>5-12</td>
</tr>
</tbody>
</table>

## Working with the GPIB-VXI Interface
<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding VISA_GPIB_VXI</td>
<td>5-22</td>
</tr>
<tr>
<td>Creating a VISA-GPIB-VXI Object</td>
<td>5-23</td>
</tr>
<tr>
<td>The VISA-GPIB-VXI Address</td>
<td>5-25</td>
</tr>
</tbody>
</table>

## Working with the Serial Port Interface
<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding the Serial Port</td>
<td>5-27</td>
</tr>
<tr>
<td>Creating a VISA-Serial Object</td>
<td>5-27</td>
</tr>
<tr>
<td>Configuring Communication Settings</td>
<td>5-29</td>
</tr>
</tbody>
</table>

## Working with the USB Interface
<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating a VISA-USB Object</td>
<td>5-31</td>
</tr>
<tr>
<td>The VISA-USB Address</td>
<td>5-33</td>
</tr>
</tbody>
</table>

## Working with the TCP/IP Interface
<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding VISA-TCP/IP</td>
<td>5-35</td>
</tr>
<tr>
<td>Creating a VISA-TCPIP Object</td>
<td>5-35</td>
</tr>
<tr>
<td>The VISA-TCPIP Address</td>
<td>5-37</td>
</tr>
</tbody>
</table>

## Working with the RSIB Interface
<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating a VISA-RSIB Object</td>
<td>5-39</td>
</tr>
<tr>
<td>The VISA-RSIB Address</td>
<td>5-41</td>
</tr>
</tbody>
</table>
Controlling Instruments Using the Serial Port

Serial Port Overview ............................................ 6-2
  What Is Serial Communication? ........................... 6-2
  The Serial Port Interface Standard ..................... 6-3
  Supported Platforms ....................................... 6-3
  Connecting Two Devices with a Serial Cable .......... 6-3
  Serial Port Signals and Pin Assignments ............. 6-5
  Serial Data Format ......................................... 6-9
  Finding Serial Port Information for Your Platform .... 6-13

The Serial Port Object ........................................ 6-16
  Creating a Serial Port Object ............................ 6-16
  The Serial Port Object Display ......................... 6-17

Configuring Communication Settings ..................... 6-19

Writing and Reading Data ................................ 6-20
  Asynchronous Write and Read Operations .............. 6-20
  Rules for Completing Write and Read Operations .... 6-21
  Example: Writing and Reading Text Data ............... 6-22

Events and Callbacks ....................................... 6-25
  Event Types and Callback Properties .................. 6-25
  Storing Event Information ............................... 6-26
  Example: Using Events and Callbacks .................. 6-28

Using Control Pins ......................................... 6-30
  Control Pins .................................................. 6-30
  Signaling the Presence of Connected Devices .......... 6-30
  Controlling the Flow of Data: Handshaking .......... 6-33

Controlling Instruments Using TCP/IP and UDP

TCP/IP and UDP Overview .................................. 7-2
## Creating a TCP/IP Object

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP/IP Object</td>
<td>7-4</td>
</tr>
<tr>
<td>TCP/IP Object Display</td>
<td>7-5</td>
</tr>
<tr>
<td>Example: Communicating with a Remote Host</td>
<td>7-6</td>
</tr>
<tr>
<td>Example: Server Drops the Connection</td>
<td>7-7</td>
</tr>
</tbody>
</table>

## Creating an UDP Object

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP Object</td>
<td>7-10</td>
</tr>
<tr>
<td>The UDP Object Display</td>
<td>7-11</td>
</tr>
<tr>
<td>Example: Communicating Between Two Hosts</td>
<td>7-12</td>
</tr>
</tbody>
</table>

## Writing and Reading Data

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules for Completing Write and Read Operations</td>
<td>7-14</td>
</tr>
<tr>
<td>Example: Writing and Reading Data with a TCP/IP Object</td>
<td>7-15</td>
</tr>
<tr>
<td>Example: Writing and Reading Data with a UDP Object</td>
<td>7-18</td>
</tr>
</tbody>
</table>

## Events and Callbacks

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Types and Callback Properties</td>
<td>7-21</td>
</tr>
<tr>
<td>Storing Event Information</td>
<td>7-22</td>
</tr>
<tr>
<td>Example: Using Events and Callbacks</td>
<td>7-24</td>
</tr>
</tbody>
</table>

## Using Device Objects

8

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Objects</td>
<td>8-2</td>
</tr>
<tr>
<td>Device Objects: Overview</td>
<td>8-2</td>
</tr>
<tr>
<td>What Are Device Objects?</td>
<td>8-2</td>
</tr>
<tr>
<td>Device Objects for MATLAB Instrument Drivers</td>
<td>8-3</td>
</tr>
</tbody>
</table>

## Creating and Connecting Device Objects

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Objects for MATLAB Interface Drivers</td>
<td>8-5</td>
</tr>
<tr>
<td>Device Objects for VXIplug&amp;play and IVI Drivers</td>
<td>8-7</td>
</tr>
<tr>
<td>Connecting the Device Object</td>
<td>8-7</td>
</tr>
</tbody>
</table>

## Communicating with Instruments

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuring Instrument Settings</td>
<td>8-9</td>
</tr>
<tr>
<td>Calling Device Object Methods</td>
<td>8-10</td>
</tr>
</tbody>
</table>
Using VXIplug&play Drivers

Overview .......................................................... 9-2
The Instrument Control Toolbox and VXIplug&play Drivers ........................................ 9-2
VISA Setup .......................................................... 9-2
Other Software Requirements .................................. 9-3

VXIplug&play Drivers .................................................. 9-4
Installing VXI plug&play Drivers ................................. 9-4
Creating a MATLAB VXIplug&play Instrument Driver ............................................. 9-5
Constructing Device Objects Using a MATLAB VXIplug&play Instrument Driver ................ 9-8

Using IVI Drivers

IVI Drivers Overview ................................................. 10-2
Instrument Control Toolbox and IVI Drivers ...................... 10-2
VISA Setup .................................................................. 10-2
IVI Shared Components .................................................. 10-3
IVI Configuration Store Overview ...................................... 10-3

IVI Drivers ................................................................. 10-4
MATLAB IVI Driver ...................................................... 10-4
IVI-C and IVI-COM ....................................................... 10-4
Installing IVI Drivers .................................................... 10-5
MATLAB IVI Instrument Drivers ........................................ 10-6
Saving Instrument Objects to an M-File ............... 12-2
Saving Objects to a MAT-File ........................ 12-4

Debugging: Recording Information to Disk .......... 12-6
Using the record Function ............................ 12-6
Example: Introduction to Recording Information .... 12-7
Creating Multiple Record Files ...................... 12-7
Specifying a File Name .............................. 12-8
Record File Format .................................. 12-8
Example: Recording Information to Disk ............ 12-10

Test & Measurement Tool

13

Test & Measurement Tool: Overview .................. 13-2
Instrument Control Toolbox Support .................. 13-2
Navigating the Tree .................................. 13-3

Using the Test & Measurement Tool .................. 13-4
Overview of the Examples ............................ 13-4
Hardware ............................................ 13-4
Instrument Objects .................................. 13-12
Instrument Drivers ................................. 13-16

Using the Instrument Driver Editor

14

MATLAB Instrument Driver Editor: Overview ........ 14-2
How Does a MATLAB Instrument Driver Work? .... 14-3

Creating MATLAB Instrument Drivers .............. 14-5
Driver Components .................................. 14-5
MATLAB Instrument Driver Editor Features ........ 14-6
Saving MATLAB Instrument Drivers ............... 14-6
Using the Instrument Driver Testing Tool

15

  Functionality ........................................ 15-2
  Drivers ........................................... 15-2
  Test Structure ..................................... 15-3
  Starting ......................................... 15-3
  Example ......................................... 15-4

Setting Up Your Test .................................................... 15-5
  Test File ........................................... 15-5
  Providing a Name and Description .............................. 15-5
  Specifying the Driver ..................................... 15-5
  Specifying an Interface .................................... 15-5
  Setting Test Preferences .................................. 15-6
  Example: Setting Up a Driver Test ............................ 15-7
Defining Test Steps ........................................ 15-11
Test Step: Set Property .................................... 15-11
Test Step: Get Property ................................... 15-15
Test Step: Properties Sweep ............................. 15-17
Test Step: Function ...................................... 15-21

Saving Your Test ........................................... 15-25
Saving the Test as M-Code ................................. 15-25
Saving the Test as a Driver Function ...................... 15-25

Testing and Results ...................................... 15-28
Running All Steps ....................................... 15-28
Partial Testing .......................................... 15-30
Exporting Results ........................................ 15-30
Saving Results .......................................... 15-31

Using the Instrument Control Toolbox Block Library

Overview .................................................... 16-2

Opening the Instrument Control Block Library .......... 16-3
Using the instrumentlib Command from MATLAB .... 16-3
Using the Simulink Library Browser ..................... 16-4

Building Simulink Models to Send and Receive Data ... 16-6
Example: Sending and Receiving Data Through a Serial
       Port Loopback ...................................... 16-6
Example: Sending and Receiving Data Via a TCP/IP
       Network .......................................... 16-16
## Functions — By Category

<table>
<thead>
<tr>
<th>Category</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instrument Object Creation</td>
<td>17-2</td>
</tr>
<tr>
<td>Interface Object</td>
<td>17-2</td>
</tr>
<tr>
<td>Device Object</td>
<td>17-2</td>
</tr>
<tr>
<td>State Change</td>
<td>17-2</td>
</tr>
<tr>
<td>Interface Object</td>
<td>17-3</td>
</tr>
<tr>
<td>Device Object</td>
<td>17-3</td>
</tr>
<tr>
<td>Property Display and Configuration</td>
<td>17-3</td>
</tr>
<tr>
<td>Reading Data</td>
<td>17-3</td>
</tr>
<tr>
<td>Writing Data</td>
<td>17-4</td>
</tr>
<tr>
<td>Information and Help</td>
<td>17-5</td>
</tr>
<tr>
<td>Graphical Tools</td>
<td>17-5</td>
</tr>
<tr>
<td>Interface Object</td>
<td>17-5</td>
</tr>
<tr>
<td>Device Object</td>
<td>17-5</td>
</tr>
<tr>
<td>General Purpose</td>
<td>17-6</td>
</tr>
<tr>
<td>Interface Objects</td>
<td>17-7</td>
</tr>
<tr>
<td>GPIB</td>
<td>17-7</td>
</tr>
<tr>
<td>Serial Port</td>
<td>17-8</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>17-8</td>
</tr>
<tr>
<td>UDP</td>
<td>17-8</td>
</tr>
<tr>
<td>VISA-GPIB</td>
<td>17-8</td>
</tr>
<tr>
<td>VISA-GPIB-VXI</td>
<td>17-8</td>
</tr>
<tr>
<td>VISA-Serial</td>
<td>17-9</td>
</tr>
<tr>
<td>VISA-VXI</td>
<td>17-9</td>
</tr>
<tr>
<td>Device Objects</td>
<td>17-10</td>
</tr>
<tr>
<td>IVI Configuration Store Objects</td>
<td>17-10</td>
</tr>
</tbody>
</table>
## Functions — Alphabetical List

## Properties — By Category

<table>
<thead>
<tr>
<th>Interface Object Base Properties</th>
<th>19-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing Data</td>
<td>19-2</td>
</tr>
<tr>
<td>Reading Data</td>
<td>19-2</td>
</tr>
<tr>
<td>Recording Data</td>
<td>19-3</td>
</tr>
<tr>
<td>Callbacks</td>
<td>19-3</td>
</tr>
<tr>
<td>General Purpose</td>
<td>19-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Interface-Specific Properties</th>
<th>19-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIB</td>
<td>19-5</td>
</tr>
<tr>
<td>Serial Port</td>
<td>19-6</td>
</tr>
<tr>
<td>TCPIP</td>
<td>19-7</td>
</tr>
<tr>
<td>UDP</td>
<td>19-7</td>
</tr>
<tr>
<td>VISA-GPIB</td>
<td>19-8</td>
</tr>
<tr>
<td>VISA-GPIB-VXI</td>
<td>19-8</td>
</tr>
<tr>
<td>VISA-RSIB</td>
<td>19-9</td>
</tr>
<tr>
<td>VISA-Serial</td>
<td>19-10</td>
</tr>
<tr>
<td>VISA-TCPIP</td>
<td>19-10</td>
</tr>
<tr>
<td>VISA-USB</td>
<td>19-11</td>
</tr>
<tr>
<td>VISA-VXI</td>
<td>19-11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device Object Base Properties</th>
<th>19-13</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>IVI Configuration Store Object Properties</th>
<th>19-14</th>
</tr>
</thead>
</table>
Getting Started

What Is Instrument Control Toolbox? (p. 1-2)
The toolbox and the kinds of tasks it can perform

Toolbox Components (p. 1-5)
The M-files and interface driver adaptors that compose the toolbox

Understanding the Toolbox Capabilities (p. 1-9)
Resources to help you understand the toolbox capabilities including demos and documentation examples

Installation Information (p. 1-10)
How to determine whether the toolbox is installed on your system

Examining Your Hardware Resources (p. 1-12)
Return hardware-related information visible to the toolbox including the installed adaptors and the syntax for creating instrument objects.

Communicating with Your Instrument (p. 1-21)
Examples that show you how to communicate with instruments that support the GPIB, GPIB-VXI, and serial port interfaces

General Preferences for Instrument Control (p. 1-27)
MATLAB preferences related to Instrument Control Toolbox

Getting Help (p. 1-31)
Getting help using the Help browser, M-file help, and other methods
What Is Instrument Control Toolbox?

In this section...

“Getting to Know the Instrument Control Toolbox” on page 1-2
“Exploring Instrument Control Toolbox” on page 1-3
“Learning About Instrument Control Toolbox” on page 1-3
“Using the Documentation Examples” on page 1-4

Getting to Know the Instrument Control Toolbox

Instrument Control Toolbox is a collection of M-file 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 MATLAB.

- **Support for IVI, VXI plug&play, and MATLAB instrument drivers.**

- **Functions for transferring data between MATLAB and your instrument:**
  - The data can be binary (numerical) or text.
  - The transfer can be synchronous and block the MATLAB command line, or asynchronous and allow access to the MATLAB command line.

- **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.**

The MathWorks provides several related products that are especially relevant to the kinds of tasks you can perform with Instrument Control Toolbox. For more information about any of these products, see

http://www.mathworks.com/products/instrument/related.jsp
Exploring Instrument Control Toolbox

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 M-file, then modifying your copy. You can also extend the toolbox by adding your own M-files, or by using it in combination with other products such as MATLAB Report Generator or Data Acquisition Toolbox.

To use Instrument Control Toolbox, 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, VXIplug&play, or MATLAB instrument driver.
- Features of the interface associated with your instrument.

Learning About Instrument Control Toolbox

Start with this chapter, which describes how to examine your hardware resources, how to communicate with your instrument, how to get online help, and so on. Then read Chapter 2, “The Instrument Control Session”, which provides 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 Functions — Alphabetical List. If you want detailed information about a specific property, refer to Properties — Alphabetical List.
Using the Documentation Examples

The examples in this guide use specific instruments such as a Tektronix TDS 210 two-channel oscilloscope or an Agilent 33120A function generator. Additionally, the GPIB examples use a National Instruments GPIB controller and the serial port examples use the 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 documentation examples are collected in the example index, which is available through the Help browser. You should modify the examples to work with your specific hardware configuration.
Toolbox Components

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Passing Information Between MATLAB and Your Instrument” on page 1-5</td>
</tr>
<tr>
<td>“M-File Functions” on page 1-7</td>
</tr>
<tr>
<td>“Interface Driver Adaptor” on page 1-8</td>
</tr>
</tbody>
</table>

**Passing Information Between MATLAB and Your Instrument**

Instrument Control Toolbox consists of two distinct components: M-file functions and interface driver adaptors. These components allow you to pass information between MATLAB and your instrument. For example, the following diagram shows how information passes from MATLAB 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.

**M-File Functions**
To perform any task within your instrument control application, you must call M-file functions from the MATLAB environment. 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 functions, refer to Functions — Alphabetical List. You can also display the toolbox functions by typing

```
help instrument
```
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 MATLAB 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 Instrument Control Toolbox.

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

Supported Interfaces and Adaptor Names

<table>
<thead>
<tr>
<th>Interface</th>
<th>Adaptor Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIB</td>
<td>advantech, agilent, cec, contec, ics, iotech, keithley, mcc, ni</td>
</tr>
<tr>
<td>Serial port</td>
<td>N/A</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>N/A</td>
</tr>
<tr>
<td>UDP</td>
<td>N/A</td>
</tr>
<tr>
<td>VISA standard</td>
<td>agilent, ni, tek</td>
</tr>
</tbody>
</table>

As described in “Examining Your Hardware Resources” on page 1-12, you can list the supported interfaces and adaptor names with the instrhwinfo function. For a list of vendor driver requirements and limitations, refer to Appendix A, “Vendor Driver Requirements and Limitations”.
Understanding the Toolbox Capabilities

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Overview Help” on page 1-9</td>
</tr>
<tr>
<td>“Documentation Examples” on page 1-9</td>
</tr>
<tr>
<td>“Demos” on page 1-9</td>
</tr>
</tbody>
</table>

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

```matlab
help instrument
```

For the code for any function, type

```matlab
type function_name
```

**Documentation Examples**
This guide provides detailed examples that show you how to communicate with all supported interface types. These examples are collected in the example index, which is available through the Help browser.

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.

**Demos**
The toolbox includes a large collection of tutorial demos, which you can access through the Help browser Demos pane. Use the following command to open the Help browser to the toolbox demos:

```matlab
demo toolbox 'Instrument Control'
```

You do not need an instrument connected to your computer to use tutorials as they use prerecorded data.
Installation Information

In this section...

“Installation Requirements” on page 1-10
“Toolbox Installation” on page 1-10
“Hardware and Driver Installation” on page 1-11

Installation Requirements
To communicate with your instrument from the MATLAB environment, you must install these components:

- MATLAB 7
- Instrument Control Toolbox

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 1-8.

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

```
ver
```

at the MATLAB prompt. MATLAB displays information about the version of MATLAB 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 (http://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. For a list of vendor driver requirements and limitations, refer to Appendix A, “Vendor Driver Requirements and Limitations”.

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

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“instrhwinfo Function” on page 1-12</td>
</tr>
<tr>
<td>“Test &amp; Measurement Tool” on page 1-16</td>
</tr>
<tr>
<td>“Viewing the IVI Configuration Store” on page 1-18</td>
</tr>
</tbody>
</table>

**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 1-12
- “Interface Information” on page 1-13
- “Adaptor Information” on page 1-13
- “Instrument Object Information” on page 1-15
- “Installed Driver Information” on page 1-16

**General Toolbox Information**
For general information about Instrument Control Toolbox, type:

```matlab
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:

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

```matlab
out =

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

The `InstalledAdaptors` field indicates that the Measurement Computing Corporation 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`.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Adaptor Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpib</td>
<td>advantech, agilent, cec, contec, ics, iotech, keithley, mcc, ni</td>
</tr>
<tr>
<td>visa</td>
<td>agilent, ni, tek</td>
</tr>
</tbody>
</table>

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

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

```matlab
ghwinfo =
```
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.

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

To display information for the Tektronix VISA adaptor,

```matlab
vhwinf = instrhwinfo('visa', 'tek')
vhwinf =
```

```
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.

```matlab
vhwinf objetoConstructorName
ans =
```

```
'visa('tek', 'ASRL1::INSTR');'
'visa('tek', 'ASRL2::INSTR');'
'visa('tek', 'GPIBO::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.

\[
vg = \text{visa('tek','GPIB0::1::INSTR')};
\]

**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 1-13), type:

\[
\text{gwhinfo} = \text{instrhwinfo(g)}
\]

\[
gwhinfo =
\begin{align*}
\text{AdaptorDllName: } & [1x82 \text{ char}] \\
\text{AdaptorDllVersion: } & 'Version 2.0 (R14)' \\
\text{AdaptorName: } & 'NI' \\
\text{VendorDllName: } & 'gpib-32.dll' \\
\text{VendorDriverDescription: } & 'NI-488'
\end{align*}
\]

To display information for the VISA-GPIB object created in the (“Adaptor Information” on page 1-13), type:

\[
\text{vghwhinfo} = \text{instrhwinfo(vg)}
\]

\[
vghwhinfo =
\begin{align*}
\text{AdaptorDllName: } & [1x83 \text{ char}] \\
\text{AdaptorDllVersion: } & 'Version 2.0 (R14)' \\
\text{AdaptorName: } & 'TEK' \\
\text{VendorDllName: } & 'visa32.dll' \\
\text{VendorDriverDescription: } & 'Tektronix VISA Driver' \\
\text{VendorDriverVersion: } & 2.0500
\end{align*}
\]

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:

```plaintext
instrhwinfo('ivi')
an =
    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 VXIplug&play` driver:

```plaintext
instrhwinfo('vxipnp', 'ag3325b')
an =
    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:

```plaintext
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 VXIplug&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 1-18.
The Test & Measurement Tool GUI includes embedded help. For further details about the Test & Measurement Tool and its capabilities, see Chapter 13, “Test & Measurement Tool”.

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 10-11.

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.

```matlab
instrhwinfo('ivi')
ans =
    LogicalNames: {'MainScope', 'FuncGen'}
    ProgramIDs: {'TekScope.TekScope','Agilent33250'}
    Modules: {'ag3325b', 'hpe363xa'}
    ConfigurationServerVersion: '1.3.1.0'
```
Examining Your Hardware Resources

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.

\[
\text{instrhwinfo('ivi','MainScope')} \\
\text{ans} = \\
\begin{align*}
\text{DriverSession: 'TekScope.DriverSession'} \\
\text{HardwareAsset: 'TekScope.Hardware'} \\
\text{SoftwareModule: 'TekScope.Software'} \\
\text{IOResourceDescriptor: 'GPIB0::13::INSTR'} \\
\text{SupportedInstrumentModels: 'TekScope 5000, 6000 and 7000 series'} \\
\text{ModuleDescription: 'TekScope software module desc'} \\
\text{ModuleLocation: ''}
\end{align*}
\]

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:

\[
\text{tmtool}
\]

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

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Instrument Control Session Examples” on page 1-21</td>
</tr>
<tr>
<td>“Example: Communicating with a GPIB Instrument” on page 1-21</td>
</tr>
<tr>
<td>“Example: Communicating with a GPIB-VXI Instrument” on page 1-22</td>
</tr>
<tr>
<td>“Example: Communicating with a Serial Port Instrument” on page 1-24</td>
</tr>
<tr>
<td>“Example: Communicating with a GPIB Instrument Using a Device Object” on page 1-25</td>
</tr>
</tbody>
</table>

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.

If you want detailed information about any functions that are used, refer to Functions — By Category. If you want detailed information about any properties that are used, refer to Properties — By Category.

Example: 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 an Agilent 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 Chapter 4, “Controlling Instruments Using GPIB”.

1-21
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.

\[
g = 	ext{gpib('ni',0,1)};
\]

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.

\[
\text{set}(g,'EOSMode','read&write')
\]

\[
\text{set}(g,'EOSCharCode','LF')
\]

4 Write and read data — Change the instrument’s peak-to-peak voltage to three volts by writing the \texttt{Volt 3} command, query the peak-to-peak voltage value, and then read the voltage value.

\[
\text{fprintf}(g,'Volt 3')
\]

\[
\text{fprintf}(g,'Volt?')
\]

\[
data = \text{fscanf}(g)
\]

\[
data = +3.00000E+00
\]

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.

\[
fclose(g)
\]

\[
delete(g)
\]

\[
clear g
\]

Example: 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 Agilent Technologies.
The GPIB controller is an Agilent E1406A command module in VXI slot 0. The instrument is an Agilent 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 Chapter 5, “Controlling Instruments Using VISA”.

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.

\[
v = \text{visa('agilent','GPIB-VXI0::80::INSTR')};
\]

2 **Connect to the instrument** — Connect \( v \) to the instrument.

\[
\text{fopen}(v)
\]

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

\[
\text{set}(v, 'EOSMode', 'read')
\]
\[
\text{set}(v, 'EOSCharCode', 'LF')
\]

4 **Write and read data** — Change the instrument’s peak-to-peak voltage to three volts by writing the \texttt{Volt 3} command, query the peak-to-peak voltage value, and then read the voltage value.

\[
\text{fprintf}(v, 'Volt 3')
\]
\[
\text{fprintf}(v, 'Volt?')
\]
\[
data = \text{fscanf}(v)
\]
\[
data = +3.00000E+00
\]

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.

\[
\text{fclose}(v)
\]
\[
\text{delete}(v)
\]
Getting Started

Example: 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 COM1 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 Chapter 6, “Controlling Instruments Using the Serial Port”.

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

\[ s = \text{serial('COM1')}; \]

2 Configure property values — Configure s to match the instrument’s baud rate and terminator.

\[ \text{set}(s, 'BaudRate', 4800) \]
\[ \text{set}(s, 'Terminator', 'CR') \]

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.

\[ \text{fopen}(s) \]

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.

\[ \text{fprintf}(s, '*IDN?') \]
\[ \text{out} = \text{fscanf}(s) \]
\[ \text{out} = \]
\[ \text{TEKTRONIX, TDS 210, 0, CF: 91.1CT FV: v1.16 TDS2CM: CMV: v1.04} \]
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.

```matlab
close(s)
delete(s)
clear s
```

**Example: 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 Keithley card, and the instrument is an Agilent 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 Chapter 8, “Using Device Objects”.

1 **Create instrument objects** — Create the GPIB object `g` associated with a Keithley 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`.

```matlab
g = gpib('keithley',0,4);
d = icdevice('agilent_33120a.mdd',g);
```

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

```matlab
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.

```matlab
devicereset(d)
```

4 **Configure property values** — Configure `d` to set the amplitude and frequency for the signal from the function generator.
set(d,'Amplitude',1.00,'AmplitudeUnits','vpp')
set(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 1-27
“MATLAB Instrument Driver Editor” on page 1-28
“MATLAB Instrument Driver Testing Tool” on page 1-28
“Device Objects” on page 1-29
“IVI Configuration Store” on page 1-30

Accessing General Preferences

You access the general preferences from your MATLAB desktop environment by selecting File > Preferences. The Preferences dialog box opens. There are two options listed for Instrument Control. Under the General node, select Confirmation Dialogs in the tree.
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** node in the tree.

The **Device Objects** section of the dialog box contains preferences related to the construction and use of device objects for VXIplug&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 & Measurement Tool (tmtool).

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.
Getting Help

In this section...

“instrhelp Function” on page 1-31
“propinfo Function” on page 1-31
“Online Support” on page 1-33

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. For example, to display all functions and properties associated with a GPIB object, as well as the constructor M-file help, 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('agilent','GPIB-VXI::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 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:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>The property data type. Possible values are any, ASCII value, callback, double, string, and struct.</td>
</tr>
<tr>
<td>Constraint</td>
<td>The type of constraint on the property value. Possible values are ASCII value, bounded, callback, enum, and none.</td>
</tr>
<tr>
<td>ConstraintValue</td>
<td>The property value constraint. The constraint can be a range of values or a list of string values.</td>
</tr>
<tr>
<td>DefaultValue</td>
<td>The property default value.</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>The condition under which a property is read only. Possible values are always, never, whileOpen, and whileRecording.</td>
</tr>
<tr>
<td>InterfaceSpecific</td>
<td>If the property is interface-specific, a 1 is returned. If the property is supported for all interfaces, a 0 is returned.</td>
</tr>
</tbody>
</table>

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

```matlab
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 string.
- 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,

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

**Online Support**

For online support of Instrument Control Toolbox, visit the Web site [http://www.mathworks.com/support/product/IC/](http://www.mathworks.com/support/product/IC/). This site includes documentation, examples, solutions, downloads, system requirements, and contact information.
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 (p. 2-2)**: Create a MATLAB object that represents the interface or the instrument.
- **Connecting to the Instrument (p. 2-4)**: Establish a connection between the object and the interface or instrument.
- **Configuring and Returning Properties (p. 2-5)**: Read and write property values to configure the interface and instrument settings.
- **Communicating with Your Instrument (p. 2-7)**: Write commands to the instrument, read data from the instrument, or call instrument driver functions.
- **Disconnecting and Cleaning Up (p. 2-8)**: Disconnect the object, and remove the object from memory and from the workspace.
- **Summary (p. 2-9)**: Choose whether to use interface objects or device objects for your application.
Creating Instrument Objects

In this section...

| “Overview” on page 2-2 |
| “Interface Objects” on page 2-2 |
| “Device Objects” on page 2-3 |

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 toolbox supports two types of instrument objects:

- Interface 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 — 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, you call the constructor for the type of interface (gpib, serial, tcpip, udp, or visa), 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 Chapter 3, “Using Interface Objects”.
**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 Chapter 8, “Using Device Objects”.
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.

```matlab
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 Properties — Alphabetical List.
Configuring and Returning Properties

In this section...

“Configuring Property Names and Property Values” on page 2-5
“Returning Property Names and Property Values” on page 2-5
“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 string values, set returns these values.
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.
Communicating with Your Instrument

In this section...

“Interface Objects and Instrument Commands” on page 2-7
“Device Objects and Instrument Drivers” on page 2-7

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 Chapter 14, “Using the Instrument Driver Editor”.

Disconnecting and Cleaning Up

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Disconnecting an Instrument Object” on page 2-8</td>
</tr>
<tr>
<td>“Cleaning Up the MATLAB Environment” on page 2-8</td>
</tr>
</tbody>
</table>

**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.

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

**Cleaning Up the MATLAB Environment**

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

```matlab
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.

```matlab
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 MATLAB with the `instrfind` function.
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 VXIplug&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 VXIplug&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 Instrument Control Toolbox.

• 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.
Using Interface Objects

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

Creating an Interface Object (p. 3-2) Create a MATLAB object that represents the instrument.

Connecting to the Instrument (p. 3-5) Establish a connection between the object and the instrument.

Configuring and Returning Properties (p. 3-6) Define the instrument object behavior by assigning values to properties.

Writing and Reading Data (p. 3-12) Write data to the instrument and read data from the instrument.

Disconnecting and Cleaning Up (p. 3-25) Disconnect the object from the instrument, and remove the object from memory and from the workspace.
Creating an Interface Object

In this section...

“Object Creation Functions” on page 3-2
“Configuring Properties During Object Creation” on page 3-3
“Creating an Array of Instrument Objects” on page 3-3

Object Creation Functions

To create an interface object, you call M-file functions called object creation functions (or object constructors). These M-files are implemented using the MATLAB object-oriented programming capabilities, which are described in “Classes and Objects” in the MATLAB documentation.

Interface Object Creation Functions

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpib</td>
<td>Create a GPIB object.</td>
</tr>
<tr>
<td>serial</td>
<td>Create a serial port object.</td>
</tr>
<tr>
<td>tcpip</td>
<td>Create a TCPIP object.</td>
</tr>
<tr>
<td>udp</td>
<td>Create a UDP object.</td>
</tr>
<tr>
<td>visa</td>
<td>Create a VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, or VISA-serial object.</td>
</tr>
</tbody>
</table>

You can find out how to create an interface object for a particular interface and adaptor with the ObjectConstructorName field of the instrhwinfo function. For example, to find out how to create a GPIB object for a National Instruments GPIB controller,

```matlab
out = instrhwinfo('gpib','ni');
out.ObjectConstructorName
ans =
    'gpib('ni', 0, 1);'
```
Configuring Properties During Object Creation

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 “Configuring and Returning Properties” on page 3-6, you configure properties using the `set` function or the dot notation. You can also configure properties during object creation by specifying property name/property value pairs. For example, the following command configures the `EOSMode` and `EOSCharCode` properties for the GPIB object `g`:

```matlab
g = gpib('ni',0,1,'EOSMode','read','EOSCharCode','CR');
```

If you specify an invalid property name or property value, the object is not created. For detailed property descriptions, refer to Properties — Alphabetical List.

Creating an Array of Instrument Objects

In MATLAB, you can create an array from existing variables by concatenating those variables. The same is true for instrument objects. For example, suppose you create the GPIB objects `g1` and `g2`:

```matlab
g1 = gpib('ni',0,1);
g2 = gpib('ni',0,2);
```

You can now create an instrument object array consisting of `g1` and `g2` using the usual MATLAB syntax. To create the row array `x`:

```matlab
x = [g1 g2]
```

Instrument Object Array

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
<th>Status</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>gpib</td>
<td>closed</td>
<td>GPIB0-1</td>
</tr>
<tr>
<td>2</td>
<td>gpib</td>
<td>closed</td>
<td>GPIB0-2</td>
</tr>
</tbody>
</table>

To create the column array `y`:

```matlab
y = [g1;g2];
```
Note that you cannot create a matrix of instrument objects. For example, you cannot create the matrix

\[
z = [g1 \; g2; g1 \; g2];
\]

??? Error using ==> gpib/vertcat
Only a row or column vector of instrument objects can be created.

Depending on your application, you might want to pass an array of instrument objects to a function. For example, using one call to the set function, you can configure both g1 and g2 to the same property value.

\[
set(x, 'EOSMode', 'read')
\]

Refer to Functions — Alphabetical List to see which functions accept an instrument object array as an input argument.
Connecting to the Instrument

Before you can use the instrument object to write or read data, you must connect it to the instrument whose address or port is specified in the creation function. You connect an interface object to the instrument with the `fopen` function.

\[ \text{fopen}(g) \]

Some properties are read-only while the object is connected and must be configured before using `fopen`. Examples include the `InputBufferSize` and the `OutputBufferSize` properties. You can determine when a property is configurable with the `propinfo` function, or by referring to Properties — Alphabetical List.

**Note** You can create any number of instrument objects. However, at any time, you can connect only one instrument object to an instrument with a given address or port.

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

\[ g.\text{Status} \]
\[ \text{ans} = \]
\[ \text{open} \]

As illustrated below, the connection between the instrument object and the instrument is complete, and you can write and read data.
Configuring and Returning Properties

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Base and Interface-Specific Properties” on page 3-6</td>
</tr>
<tr>
<td>“Returning Property Names and Property Values” on page 3-6</td>
</tr>
<tr>
<td>“Configuring Property Values” on page 3-9</td>
</tr>
<tr>
<td>“Specifying Property Names” on page 3-10</td>
</tr>
<tr>
<td>“Default Property Values” on page 3-10</td>
</tr>
<tr>
<td>“Property Inspector” on page 3-11</td>
</tr>
</tbody>
</table>

**Base and Interface-Specific Properties**

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**: These are supported for all interface objects (serial port, GPIB, VISA-VXI, and so on). For example, the `BytesToOutput` property is supported for all interface objects.

- **Interface-Specific Properties**: These 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.

**Returning Property Names and Property Values**

Once the instrument object is created, you can use the `set` function to return all configurable properties to a variable or to the command line. Additionally, if a property has a finite set of string values, then `set` also returns these values.
For example, the configurable properties for the GPIB object `g` are shown below. The base properties are listed first, followed by the GPIB-specific properties.

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

```matlab
>> set(g)
```

- ByteOrder: [ {littleEndian} | bigEndian ]
- BytesAvailableFcn
- BytesAvailableFcnCount
- BytesAvailableFcnMode: [ {eosCharCode} | byte ]
- ErrorFcn
- InputBufferSize
- Name
- OutputBufferSize
- OutputEmptyFcn
- RecordDetail: [ {compact} | verbose ]
- RecordMode: [ {overwrite} | append | index ]
- RecordName
- Tag
- Timeout
- TimerFcn
- TimerPeriod
- UserData

**GPIB specific properties:**
- BoardIndex
- CompareBits
- EOIMode: [ {on} | off ]
- EOSCharCode
- EOSMode: [ {none} | read | write | read&write ]
- PrimaryAddress
- SecondaryAddress

You can use the `get` function to return one or more properties and their current values to a variable or to the command line.

For example, all the properties and their current values for the GPIB object `g` are shown below. The base properties are listed first, followed by the GPIB-specific properties.

```matlab
>> get(g)
```
ByteOrder = littleEndian
BytesAvailable = 0
BytesAvailableFcn =
BytesAvailableFcnCount = 48
BytesAvailableFcnMode = eosCharCode
BytesToOutput = 0
ErrorFcn =
InputBufferSize = 512
Name = GPIB0-1
OutputBufferSize = 512
OutputEmptyFcn =
RecordDetail = compact
RecordMode = overwrite
RecordName = record.txt
RecordStatus = off
Status = closed
Tag =
Timeout = 10
TimerFcn =
TimerPeriod = 1
TransferStatus = idle
Type = gpib
UserData = []
ValuesReceived = 0
ValuesSent = 0

GPIB specific properties:
BoardIndex = 0
BusManagementStatus = [1x1 struct]
CompareBits = 8
EOIMode = on
EOSCharCode = LF
EOSMode = none
HandshakeStatus = [1x1 struct]
PrimaryAddress = 1
SecondaryAddress = 0

To display the current value for one property, you supply the property name to get.
Configuring and Returning Properties

```matlab
get(g,'OutputBufferSize')
ans =
   512
```

To display the current values for multiple properties, you include the property names as elements of a cell array.

```matlab
get(g,{'BoardIndex','TransferStatus'})
ans =
   0    'idle'
```

You can also use the dot notation to display a single property value.

```matlab
g.PrimaryAddress
ans =
   1
```

**Configuring Property Values**

You can configure property values using the `set` function

```matlab
set(g,'EOSMode','read')
```

or the dot notation.

```matlab
g.EOSMode = 'read';
```

To configure values for multiple properties, you can supply multiple property name/property value pairs to `set`.

```matlab
set(g,'EOSCharCode','CR','Name','Test1-gpib')
```

Note that you can configure only one property value at a time using the dot notation.

In practice, you can configure many of the properties at any time while the instrument object exists — including during object creation. However, some properties are not configurable while the object is connected to the instrument or when recording information to disk. Use the `propinfo` function, or refer to Properties — Alphabetical List to understand when you can configure a property.
Specifying Property Names

Instrument object property names are presented using mixed case. While this makes property names easier to read, you can use any case you want when specifying property names. Additionally, you need use only enough letters to identify the property name uniquely, so you can abbreviate most property names. For example, you can configure the EOSMode property in any of these ways.

```matlab
set(g, 'EOSMode', 'read')
set(g, 'eosmode', 'read')
set(g, 'EOSM', 'read')
```

However, when you include property names in an M-file, you should use the full property name. This practice can prevent problems with future releases of Instrument Control Toolbox if a shortened name is no longer unique because of the addition of new properties.

Default Property Values

If you do not explicitly define a value for a property, then the default value is used. All configurable properties have default values.

**Note**

Default values are provided for all instrument object properties. For serial port objects, the default values are provided by your operating system. For GPIB and VISA instrument objects, the default values are provided by vendor-supplied tools. However, these settings are overridden by your MATLAB code, and will have no effect on your instrument control application.

If a property has a finite set of string values, then the default value is enclosed by `{}` (curly braces). For example, the default value for the EOSMode property is none.

```matlab
set(g, 'EOSMode')
[ {none} | read | write | read&write ]
```

You can also use the `propinfo` function, or refer to Functions — Alphabetical List to find the default value for any property.
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.
Writing and Reading Data

In this section...

| “Before Performing Read/Write Operations” on page 3-12 |
| “Writing Data” on page 3-13 |
| “Reading Data” on page 3-19 |

Before Performing Read/Write 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, you should consider these three questions:

- What is the process by which data flows from MATLAB to the instrument, and from the instrument to MATLAB?

  Instrument Control Toolbox automatically manages the data transferred between MATLAB 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, executing an asynchronous read or write operation, or debugging your application, you might need to be aware of how this process works.

- Is the data to be transferred binary (numerical) or text (ASCII)?

  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.

- Will the write or read function block access to the MATLAB command line?

  You control access to the MATLAB command line by specifying whether a read or write operation is synchronous or asynchronous. A synchronous operation blocks access to the command line until the read or write function completes execution. An asynchronous operation does not block access to
the command, and you can issue additional commands while the read or write function executes in the background.

There are other issues to consider when you read and write data, like the conditions under which a read or write operation completes. These issues vary depending upon the supported interface and are described in the respective interface-specific chapters.

Writing Data

Functions Associated with Writing Data

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binblockwrite</td>
<td>Write binblock data to the instrument.</td>
</tr>
<tr>
<td>fprintf</td>
<td>Write text to the instrument.</td>
</tr>
<tr>
<td>fwrite</td>
<td>Write binary data to the instrument.</td>
</tr>
<tr>
<td>stopasync</td>
<td>Stop asynchronous read and write operations.</td>
</tr>
</tbody>
</table>

Properties Associated with Writing Data

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesToOutput</td>
<td>Indicate the number of bytes currently in the output buffer.</td>
</tr>
<tr>
<td>OutputBufferSize</td>
<td>Specify the size of the output buffer in bytes.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Specify the waiting time to complete a read or write operation.</td>
</tr>
</tbody>
</table>
Properties Associated with Writing Data (Continued)

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransferStatus</td>
<td>Indicate if an asynchronous read or write operation is in progress.</td>
</tr>
<tr>
<td>ValuesSent</td>
<td>Indicate the total number of values written to the instrument.</td>
</tr>
</tbody>
</table>

Output Buffer and Data Flow

The output buffer is computer memory allocated by the instrument object to store data that is to be written to the instrument. The flow of data from MATLAB to your instrument follows these steps:

1. The data specified by the write function is sent to the output buffer.
2. The data in the output buffer is sent to the instrument.

The OutputBufferSize property specifies the maximum number of bytes that you can store in the output buffer. The BytesToOutput property indicates the number of bytes currently in the output buffer. The default values for these properties are:

```matlab
    g = gpib('ni',0,1);
    get(g,{'OutputBufferSize','BytesToOutput'})
    ans =
          [512]    [0]
```

If you attempt to write more data than can fit in the output buffer, an error is returned and no data is written.

**Note** When writing data, you might need to specify a value, which can consist of one or more bytes. This is because some write functions allow you to control the number of bits written for each value and the interpretation of those bits as character, integer or floating-point values. For example, if you write one value from an instrument using the int32 format, then that value consists of four bytes.
For example, suppose you write the string command *IDN? to an instrument using the fprintf function. As shown below, the string is first written to the output buffer as six values.

### MATLAB

```matlab
G = gpib('ni', 0, 1); G.EOSMode = 'write'; fopen(G); fprintf(G, '*IDN?')
```

### Output Buffer

*IDN? six values six bytes

<table>
<thead>
<tr>
<th>Bytes used during write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes unused during write</td>
</tr>
</tbody>
</table>

The *IDN? command consists of six values because the End-Of-String character is written to the instrument, as specified by the EOSMode and EOSCharCode properties. Moreover, the default data format for the fprintf function specifies that one value corresponds to one byte.

As shown below, after the string is stored in the output buffer, it is then written to the instrument.

### Output Buffer  GPIB Board  Instrument

*IDN? six values six bytes

<table>
<thead>
<tr>
<th>Bytes used during write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes unused during write</td>
</tr>
</tbody>
</table>

**Example: 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 `fprintf` function. By default, `fprintf` uses the `%s
` format, which formats the data as a string and includes the terminator. You can write binary data with the `fwrite` function. By default, `fwrite` writes data using the `uchar` precision, which translates the data as unsigned 8-bit characters. Both of these functions support many other formats and precisions, as described in their reference pages.

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 GPIB object `g` associated with a National Instruments GPIB controller with board index 0, and an instrument with primary address 1. The size of the output buffer is increased to accommodate the waveform data. You must configure the `OutputBufferSize` property while the GPIB object is disconnected from the instrument.

   ```matlab
   g = gpib('ni',0,1);
g.OutputBufferSize = 3000;
   ```

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

   ```matlab
   fopen(g)
   ```

3 Write and read data — Write string commands that configure the scope to store binary waveform data in memory location A.

   ```matlab
   fprintf(g,'DATA:DESTINATION REFA');
   fprintf(g,'DATA:ENCDG SRPbinary');
   fprintf(g,'DATA:WIDTH 1');
   fprintf(g,'DATA:START 1');
   ```

Create the waveform data.

   ```matlab
   t = linspace(0,25,2500);
data = round(sin(t)*90 + 127);
```
Write the binary waveform data to the scope.

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

The `ValuesSent` property indicates the total number of values that were written to the instrument.

```matlab
g.ValuesSent
ans =
    2577
```

**4 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.

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

**Synchronous Versus Asynchronous Write Operations**

By default, all write functions operate synchronously and block the MATLAB command line until the operation completes. To perform an asynchronous write operation, you supply the `async` input argument to the `fprintf` or `fwrite` function.

For example, you use the following syntax to modify the `fprintf` commands used in the preceding example to write text data asynchronously.

```matlab
fprintf(g,'DATA:DESTINATION REFA','async');
```

Similarly, you use the following syntax to modify the `fwrite` command used in the preceding example to write binary data asynchronously.

```matlab
fwrite(g,[cmd data],'async');
```

You can monitor the status of the asynchronous write operation with the `TransferStatus` property. A value of `idle` indicates that no asynchronous operations are in progress.

```matlab
g.TransferStatus
```
ans = write

You can use the `BytesToOutput` property to indicate the numbers of bytes that exist in the output buffer waiting to be written to the instrument.

```matlab
g.BytesToOutput
ans =
    2512```
Reading Data

Functions Associated with Reading Data

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binblockread</td>
<td>Read binblock data from the instrument.</td>
</tr>
<tr>
<td>fgetl</td>
<td>Read one line of text from the instrument and discard the terminator.</td>
</tr>
<tr>
<td>fgets</td>
<td>Read one line of text from the instrument and include the terminator.</td>
</tr>
<tr>
<td>fread</td>
<td>Read binary data from the instrument.</td>
</tr>
<tr>
<td>fscanf</td>
<td>Read data from the instrument, and format as text.</td>
</tr>
<tr>
<td>readasync</td>
<td>Read data asynchronously from the instrument.</td>
</tr>
<tr>
<td>scanstr</td>
<td>Read data from the instrument, format as text, and parse</td>
</tr>
<tr>
<td>stopasync</td>
<td>Stop asynchronous read and write operations.</td>
</tr>
</tbody>
</table>

Properties Associated with Reading Data

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesAvailable</td>
<td>Indicate the number of bytes available in the input buffer.</td>
</tr>
<tr>
<td>InputBufferSize</td>
<td>Specify the size of the input buffer in bytes.</td>
</tr>
<tr>
<td>ReadAsyncMode</td>
<td>Specify whether an asynchronous read is continuous or manual (serial port,</td>
</tr>
<tr>
<td></td>
<td>TCPIP, UDP, and VISA-serial objects only).</td>
</tr>
<tr>
<td>Timeout</td>
<td>Specify the waiting time to complete a read or write operation.</td>
</tr>
</tbody>
</table>
### Properties Associated with Reading Data (Continued)

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransferStatus</td>
<td>Indicate if an asynchronous read or write operation is in progress.</td>
</tr>
<tr>
<td>ValuesReceived</td>
<td>Indicate the total number of values read from the instrument.</td>
</tr>
</tbody>
</table>

### Input Buffer and Data Flow

The input buffer is computer memory allocated by the instrument object to store data that is to be read from the instrument. The flow of data from your instrument to MATLAB follows these steps:

1. The data read from the instrument is stored in the input buffer.
2. The data in the input buffer is returned to the MATLAB variable specified by the `read` function.

The `InputBufferSize` property specifies the maximum number of bytes that you can store in the input buffer. The `BytesAvailable` property indicates the number of bytes currently available to be read from the input buffer. The default values for these properties are:

```matlab
  g = gpib('ni',0,1);
  get(g,{'InputBufferSize','BytesAvailable'})
  ans =
      [512]    [0]
```

If you attempt to read more data than can fit in the input buffer, an error is returned and no data is read.

For example, suppose you use the `fscanf` function to read the text-based response of the `*IDN?` command previously written to the instrument. The data is first read into the input buffer.
Writing and Reading Data

Note that for a given read operation, you might not know the number of bytes returned by the instrument. Therefore, you might need to preset the `InputBufferSize` property to a sufficiently large value before connecting the instrument object.

As shown below, after the data is stored in the input buffer, it is then transferred to the output variable specified by `fscanf`.

**Example: 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 `fgetl`, `fgets`, and `fscanf` functions. By default, these functions return data using the `%c` format. You can read binary data with the `fread` function. By default, `fread` returns numerical values as...
double-precision arrays. Both the `fscanf` and `fread` functions support many other formats and precisions, as described in their reference pages.

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 GPIB object `g` associated with a National Instruments GPIB controller with board index 0, and an instrument with primary address 1.

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

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

   ```matlab
   fopen(g)
   ```

3 **Write and read data** — Write the *IDN? command to the scope, and read back the identification information as text.

   ```matlab
   fprintf(g,'*IDN?')
   idn = fscanf(g)
   idn =
   TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
   ```

Configure the scope to return the period of the input signal, and then read the period as a binary value. The output display format is configured to use short exponential notation for doubles.

```matlab
fprintf(g,'MEASUREMENT:MEAS1:TYPE PERIOD')
fprintf(g,'MEASUREMENT:MEAS1:VALUE?')
format short e
period = fread(g,9)
period =
   49   46   48   48   54   69   45   51   10
```

`period` consists of positive integers representing character codes, where 10 is a line feed. To convert the voltage value to a string, use the `char` function.

```matlab
char(period)
ans =
   1.006E-3
```
The `ValuesReceived` property indicates the total number of values that were read from the instrument.

```matlab
g.ValuesReceived
ans =
   65
```

4 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.

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

**Synchronous Versus Asynchronous Read Operations**

The `fgetl`, `fgets`, `fscanf`, and `fread` functions operate synchronously and block the MATLAB command line until the operation completes. To perform an asynchronous read operation, you use the `readasync` function. `readasync` asynchronously reads data from the instrument and stores it in the input buffer. To transfer the data from the input buffer to a MATLAB variable, you use one of the synchronous read functions.

**Note** For serial port, TCPIP, UDP, and VISA-serial objects, you can also perform an asynchronous read operation by configuring the `ReadAsyncMode` property to `continuous`.

For example, to modify the preceding example to asynchronously read the scope’s identification information, you would issue the `readasync` function after the `*IDN?` command is written.

```matlab
fprintf(g,'*IDN?')
readasync(g)
```

You can monitor the status of the asynchronous read operation with the `TransferStatus` property. A value of `idle` indicates that no asynchronous operations are in progress.
g.TransferStatus
ans =
read

You can use the BytesAvailable property to indicate the number of bytes that exist in the input buffer waiting to be transferred to MATLAB.

g.BytesAvailable
ans =
56

When the read completes, you can transfer the data as text to a MATLAB variable using the fscanf function.

idn = fscanf(g);
Disconnecting and Cleaning Up

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Disconnecting an Instrument Object” on page 3-25</td>
</tr>
<tr>
<td>“Cleaning Up the MATLAB Environment” on page 3-25</td>
</tr>
</tbody>
</table>

**Disconnecting an Instrument Object**

When you no longer need an instrument object, you should disconnect it from the instrument, and clean up the MATLAB environment by removing the object from memory and from the workspace.

To disconnect your communication with the instrument, use the `fclose` function.

```
fclose(g)
```

You can examine the `Status` property to verify that the object and the instrument are disconnected.

```
g.Status
ans =
closed
```

After `fclose` is issued, the resources associated with `g` are made available, and you can once again connect an instrument object to the instrument with `fopen`.

**Cleaning Up the MATLAB Environment**

To remove the instrument object from memory, use the `delete` function.

```
delete(g)
```

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 g
```
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 MATLAB with the `instrfind` function.
# Controlling Instruments Using GPIB

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIB Overview (p. 4-3)</td>
<td>Basic features of the General Purpose Interface Bus (GPIB).</td>
</tr>
<tr>
<td>Creating a GPIB Object (p. 4-15)</td>
<td>The GPIB object establishes a connection between MATLAB and the instrument via its GPIB interface.</td>
</tr>
<tr>
<td>Configuring the GPIB Address (p. 4-18)</td>
<td>The GPIB address consists of the board index of the GPIB controller, and the primary address and (optionally) the secondary address of the instrument.</td>
</tr>
<tr>
<td>Writing and Reading Data (p. 4-19)</td>
<td>Interface-specific issues related to writing and reading data with a GPIB object.</td>
</tr>
<tr>
<td>Events and Callbacks (p. 4-28)</td>
<td>Enhance your instrument control application using events and callbacks.</td>
</tr>
</tbody>
</table>
Triggers (p. 4-36)  
Send the GET (Group Execute Trigger) GPIB command to the instrument. This command instructs all addressed Listeners to perform a specified action.

Serial Polls (p. 4-39)  
Execute a serial poll where 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.
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. The 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 GPIB 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 “Creating a GPIB Object” on page 4-15. If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to Chapter 2, “The Instrument Control Session”.
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 below. For detailed information about GPIB functionality, see the appropriate references in the Appendix B, “Bibliography”.

**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. There can be only one Talker at a given time.
- Listeners — A Listener receives data over the interface when addressed to listen by the Controller. There can be 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**

There are two types of data that 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 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 below.

**GPIB Pin and Signal Assignments**

<table>
<thead>
<tr>
<th>Pin</th>
<th>Label</th>
<th>Signal Name</th>
<th>Pin</th>
<th>Label</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DIO1</td>
<td>Data transfer</td>
<td>13</td>
<td>DIO5</td>
<td>Data transfer</td>
</tr>
<tr>
<td>2</td>
<td>DIO2</td>
<td>Data transfer</td>
<td>14</td>
<td>DIO6</td>
<td>Data transfer</td>
</tr>
<tr>
<td>3</td>
<td>DIO3</td>
<td>Data transfer</td>
<td>15</td>
<td>DIO7</td>
<td>Data transfer</td>
</tr>
<tr>
<td>4</td>
<td>DIO4</td>
<td>Data transfer</td>
<td>16</td>
<td>DIO8</td>
<td>Data transfer</td>
</tr>
<tr>
<td>5</td>
<td>EOI</td>
<td>End Or Identify</td>
<td>17</td>
<td>REN</td>
<td>Remote Enable</td>
</tr>
<tr>
<td>6</td>
<td>DAV</td>
<td>Data Valid</td>
<td>18</td>
<td>GND</td>
<td>DAV ground</td>
</tr>
<tr>
<td>7</td>
<td>NRFD</td>
<td>Not Ready For Data</td>
<td>19</td>
<td>GND</td>
<td>NRFD ground</td>
</tr>
<tr>
<td>8</td>
<td>NDAC</td>
<td>Not Data Accepted</td>
<td>20</td>
<td>GND</td>
<td>NDAC ground</td>
</tr>
<tr>
<td>9</td>
<td>IFC</td>
<td>Interface Clear</td>
<td>21</td>
<td>GND</td>
<td>IFC ground</td>
</tr>
<tr>
<td>10</td>
<td>SRQ</td>
<td>Service Request</td>
<td>22</td>
<td>GND</td>
<td>SRQ ground</td>
</tr>
</tbody>
</table>
GPIB Pin and Signal Assignments (Continued)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Label</th>
<th>Signal Name</th>
<th>Pin</th>
<th>Label</th>
<th>Signal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>ATN</td>
<td>Attention</td>
<td>23</td>
<td>GND</td>
<td>ATN ground</td>
</tr>
<tr>
<td>12</td>
<td>Shield</td>
<td>Chassis ground</td>
<td>24</td>
<td>GND</td>
<td>Signal ground</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATN</td>
<td>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.</td>
</tr>
<tr>
<td>IFC</td>
<td>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.</td>
</tr>
<tr>
<td>REN</td>
<td>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.</td>
</tr>
</tbody>
</table>
GPIB Interface Management Lines (Continued)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRQ</td>
<td>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.</td>
</tr>
<tr>
<td>EOI</td>
<td>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).</td>
</tr>
</tbody>
</table>

You can examine the state of the interface management lines with the `BusManagementStatus` property.

**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.

**GPIB Handshake Lines**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAV</td>
<td>Used by the Talker to indicate that a byte can be read by the Listeners.</td>
</tr>
<tr>
<td>NRFD</td>
<td>Indicates whether the Listener is ready to receive the byte.</td>
</tr>
<tr>
<td>NDAC</td>
<td>Indicates whether the Listener has accepted the byte.</td>
</tr>
</tbody>
</table>

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 asserts the DAV line 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 below.
You can examine the state of the handshake lines with the HandshakeStatus property.

**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 below.

**Standard Event**

![Diagram of status registers, enable registers, and output queue]

- **Standard Event Status Register**
- **Standard Event Status Enable Register**
- **Output Queue**
- **Data available on instrument ready to be sent across the bus**
- **Status Byte Register**
- **Service Request Enable Register**
- **Logical OR**
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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>–</td>
<td>Instrument-specific summary messages.</td>
</tr>
<tr>
<td>4</td>
<td>MAV</td>
<td>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.</td>
</tr>
<tr>
<td>5</td>
<td>ESB</td>
<td>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.</td>
</tr>
<tr>
<td>6</td>
<td>MSS</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>RQS</td>
<td>The Request Service bit indicates that the instrument requests service from the GPIB controller. This bit is obtained from a serial poll.</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>Instrument-specific summary message.</td>
</tr>
</tbody>
</table>

For example, if you want to know when a specific type of instrument error occurs, you would enable bit 5 of the SRER. Additionally, you would enable the appropriate bit of the Standard Event Status Enable Register (see “Standard Event Status Register” on page 4-13) 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 are described below.

Standard Event Status Register Bits

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

For example, if you want to know when an execution error occurs, you would enable bit 4 of the ESER. Additionally, you would enable bit 5 of the SRER (see “Status Byte Register” on page 4-12) 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.

GPIB Commands for Reading and Writing Register Information

<table>
<thead>
<tr>
<th>Register</th>
<th>Operation</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESR</td>
<td>Read</td>
<td>*ESR?</td>
<td>Return a decimal value that corresponds to the weighted sum of all the bits set in the SESR register.</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>N/A</td>
<td>You cannot write to the SESR register.</td>
</tr>
<tr>
<td>ESER</td>
<td>Read</td>
<td>*ESE?</td>
<td>Return a decimal value that corresponds to the weighted sum of all the bits enabled by the *ESE command.</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>*ESE</td>
<td>Write a decimal value that corresponds to the weighted sum of all the bits you want to enable in the SESR register.</td>
</tr>
<tr>
<td>SBR</td>
<td>Read</td>
<td>*STB?</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>N/A</td>
<td>You cannot write to the SBR register.</td>
</tr>
<tr>
<td>SRER</td>
<td>Read</td>
<td>*SRE?</td>
<td>Return a decimal value that corresponds to the weighted sum of all the bits enabled by the *SRE command.</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>*SRE</td>
<td>Write a decimal value that corresponds to the weighted sum of all the bits you want to enable in the SBR register.</td>
</tr>
</tbody>
</table>

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 “Example: Executing a Serial Poll” on page 4-39.
Creating a GPIB Object

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Using the gpib Function” on page 4-15</td>
</tr>
<tr>
<td>“GPIB Object Display” on page 4-16</td>
</tr>
</tbody>
</table>

**Using the gpib Function**

You create a GPIB object with the `gpib` function. `gpib` requires the adaptor name, the GPIB board index, and the primary address of the instrument. As described in “Connecting to the Instrument” on page 2-4, you can also configure property values during object creation. For a list of supported adaptors, refer to “Interface Driver Adaptor” on page 1-8.

Each GPIB object is associated with one controller and one instrument. For example, to create a GPIB object associated with a National Instruments controller with board index 0, and an instrument with primary address 1,

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

**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.

The GPIB object `g` now exists in the MATLAB workspace. You can display the class of `g` with the `whos` command.

```matlab
whos g
```

```
Name      Size    Bytes  Class
-------    -----    -----  -----
g        1x1        636  gpib object

Grand total is 14 elements using 636 bytes
```

Once the GPIB object is created, the following properties are automatically assigned values. These general-purpose properties describe the object based on its class type and address information.
**GPIB Descriptive Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the GPIB object.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for g with the get function.

```matlab
get(g,{'Name','Type'})
```

```
ans =
    'GPIB0-1'   'gpib'
```

**GPIB Object Display**

The GPIB object provides you with a convenient display that summarizes important address and state information. You can invoke the display summary as follows:

- Type the GPIB object at the command line.
- Exclude the semicolon when creating a GPIB object.
- Exclude the semicolon when configuring properties using 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.

The display summary for the GPIB object g is:

```
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
Configuring the GPIB Address

Each GPIB object is associated with one controller and one instrument. The GPIB address consists of the board index of the GPIB controller, and the primary address and (optionally) the secondary address of the instrument. The term “board index” is equivalent to the term “logical unit” as used by Agilent Technologies.

Note that some vendors place limits on the allowed board index values. Refer to Appendix A, “Vendor Driver Requirements and Limitations” for a list of these limitations. You can usually find the instrument addresses through a front panel display or by examining dip switch settings. Valid primary addresses range from 0 to 30. Valid secondary addresses range from 96 to 126, or can be 0, indicating that no secondary address is used.

The properties associated with the GPIB address are given below.

**GPIB Address Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoardIndex</td>
<td>Specify the index number of the GPIB board.</td>
</tr>
<tr>
<td>PrimaryAddress</td>
<td>Specify the primary address of the GPIB instrument.</td>
</tr>
<tr>
<td>SecondaryAddress</td>
<td>Specify the secondary address of the GPIB instrument.</td>
</tr>
</tbody>
</table>

You must specify the board index and instrument primary address values during GPIB object creation. The BoardIndex and PrimaryAddress properties are automatically updated with these values. If the instrument has a secondary address, you can specify its value during or after object creation by configuring the SecondaryAddress property.

You can display the address property values for the GPIB object \( g \) created in “Creating a GPIB Object” on page 4-15 with the get function.

```matlab
get(g,{'BoardIndex','PrimaryAddress','SecondaryAddress'})
```

```matlab
ans =
    [0]    [1]    [0]
```
Writing and Reading Data

Rules for Completing Write and Read Operations

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.

Additionally, you can stop an asynchronous write operation at any time with the stopasync function.

An instrument determines if a write operation is complete based on the EOSMode, EOIMode, and EOSCharCode property values. If EOSMode is configured to either write or read&write, each occurrence of \n in a text command is replaced with the End-Of-String (EOS) character specified by the EOSCharCode value. Therefore, when you use the default fprintf format of %s\n, all text commands written to the instrument will end with that value. The default EOSCharCode value is LF, which corresponds to the line feed character. The EOS character required by your instrument will be described in its documentation.

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. If EOSMode is configured to either write or
read&write, then the last byte written is the EOSCharCode value and the EOI line is asserted when the instrument receives this byte.

**Completing Read Operations**
A read operation with fgetl, fgets, fread, fscanf, or readasync completes when one of these conditions is satisfied:

- The EOI line is asserted.
- The terminator specified by the EOSCharCode property is read. This can occur only when the EOSMode property is configured to either read or read&write.
- The time specified by the Timeout property passes.
- The specified number of values is read (fread, fscanf, and readasync only).
- The input buffer is filled (if the number of values is not specified).

In addition to these rules, you can stop an asynchronous read operation at any time with the stopasync function.

**Example: Writing and Reading Text Data**
This example illustrates how to communicate with a GPIB instrument by writing and reading text data.

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

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.
   ```matlab
   g = gpib('ni',0,1);
   ```
2. **Connect to the instrument** — Connect `g` to the oscilloscope, and return the default values for the EOSMode and EOIMode properties.
Writing and Reading Data

```matlab
fopen(g)
get(g,{'EOSMode','EOIMode'})
ans =
    'none'    'on'
```

Using these property values, write operations complete when the last byte is written to the instrument, and read operations complete when the EOI line is asserted by the instrument.

3 Write and read data — Write the *IDN? command to the instrument using fprintf, and then read back the result of the command using fscanf.

```matlab
fprintf(g,'*IDN?')
idn = fscanf(g)
idn =
    TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

Determine the measurement source. Possible measurement sources include channel 1 and channel 2 of the oscilloscope.

```matlab
fprintf(g,'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(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.

```matlab
fprintf(g,'MEASUREMENT:IMMED:SOURCE CH2')
fprintf(g,'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(g)
source =
    CH2
```

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

```matlab
fprintf(g,'MEASUREMENT:MEAS1:TYPE PK2PK')
fprintf(g,'MEASUREMENT:MEAS1:VALUE?')
ptop = fscanf(g)
```
ptop = 2.0199999809E0

4 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

Example: Reading Binary Data
This example illustrates how you can download the TDS 210 oscilloscope screen display to MATLAB. The screen display data is transferred to MATLAB and 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:

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 property values — 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;

3 Connect to the instrument — Connect g to the oscilloscope.

   fopen(g)

4 Write and read data — 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')
Asynchronously transfer the data from the instrument to the input buffer.

```matlab
readasync(g)
```

Wait until the read operation completes, and then transfer the data to the MATLAB workspace as unsigned 8-bit integers.

```matlab
g.TransferStatus
ans =
idle
out = fread(g,g.BytesAvailable,'uint8');
```

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.

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

**Viewing the Bitmap Data**
To view the bitmap data, you should follow these steps:

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.

Note that the MATLAB file I/O versions of the `fopen`, `fwrite`, and `fclose` functions are used.

```matlab
fid = fopen('test1.bmp','w');
fwrite(fid,out,'uint8');
fclose(fid)
a = imread('test1.bmp','bmp');
imagesc(fliplr(a'))
```
Because the scope returns the screen display data using only two colors, an appropriate colormap is selected.

```matlab
mymap = [0 0 0; 1 1 1];
colormap(mymap)
```

The resulting bitmap image is shown below.

---

**Example: Parsing Input Data Using `scanstr`**

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

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

2. **Connect to the instrument** — Connect `g` to the oscilloscope.
Writing and Reading Data

3 Write and read data — Return identification information to separate elements of a cell array using the default delimiters.

```matlab
fprintf(g,'*IDN?');
idn = scanstr(g)
idn =
    'TEKTRONIX'
    'TDS 210'
    [ 0]
    'CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04'
```

4 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.

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

Example: Understanding EOI and EOS

This example illustrates how the EOI line and the EOS character are used to complete read and write operations, and how the EOIMode, EOSMode, and EOSCharCode properties are related to each other. In most cases, you can successfully communicate with your instrument by accepting the default values for these properties.

The default value for EOIMode is on, which means that the EOI line is asserted when the last byte is written to the instrument. The default value for EOSMode is none, which means that the EOSCharCode value is not written to the instrument, and read operations will not complete when the EOSCharCode value is read. Therefore, when you use the default values for EOIMode and EOSMode,

- Write operations complete when the last byte is written to the instrument.
- Read operations complete when the EOI line is asserted by the instrument.
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.

\[ g = \text{gpib('ni',0,1);} \]

2 Connect to the instrument — Connect g to the oscilloscope.

\[ \text{fopen(g)} \]

3 Write and read data — Configure g so that the EOI line is not asserted after the last byte is written to the instrument, and the EOS character is used to complete write operations. The default format for fprintf is \( \%s\n \), where \( \n \) is replaced by the EOS character as given by EOSCharCode.

\[
g.\text{EOIMode} = 'off';
g.\text{EOSMode} = 'write';
\text{fprintf(g,'*IDN?')}
\]

\[ \text{out} = \text{fscanf(g)} \]

\[ \text{out} = \]

\[ \text{TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04} \]

Although EOSMode is configured so that read operations will not complete after receiving the EOS character, the preceding read operation succeeded because the EOI line was asserted.

Now configure g so that the EOS character is not used to complete read or write operations. Because the EOI line is not asserted and the EOS character is not written, the instrument cannot interpret the *IDN? command and a timeout occurs.

\[
g.\text{EOSMode} = 'none';
g.\text{EOIMode} = 'write';
\text{fprintf(g,'*IDN?')}
\]

\[ \text{out} = \text{fscanf(g)} \]

\[ \text{Warning: GPIB: NI: An I/O operation has been canceled mostly likely due to a timeout.} \]

Now configure g so that the read operation terminates after the “X” character is read. The EOIMode property is configured to on so that the EOI line is asserted after the last byte is written. The EOSMode property
is configured to read so that the read operation completes when the EOSCharCode value is read.

```matlab
    g.EOIMode = 'on';
    g.EOSMode = 'read';
    g.EOSCharCode = 'X';
    fprintf(g,'*IDN?')
    out = fscanf(g)
    out =

    TEKTRONIX
```

Note that the rest of the identification string remains in the instrument’s hardware buffer. If you do not want to return this data during the next read operation, you should clear it from the instrument buffer with the clrdevice function.

```matlab
    clrdevice(g)
```

4 **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.

```matlab
    fclose(g)
    delete(g)
    clear g
```
Events and Callbacks

In this section...

“Example: Introduction to Events and Callbacks” on page 4-28
“Event Types and Callback Properties” on page 4-29
“Storing Event Information” on page 4-30
“Creating and Executing Callback Functions” on page 4-32
“Enabling Callback Functions After They Error” on page 4-33
“Example: Using Events and Callbacks to Read Binary Data” on page 4-33

Example: Introduction to Events and Callbacks
You can enhance the power and flexibility of your instrument control application by using events. An event occurs after a condition is met, and might result in one or more callbacks.

While the instrument object is connected to the instrument, you can use events to display a message, display data, analyze data, and so on. Callbacks are controlled through callback properties and callback functions. All event types have an associated callback property. Callback functions are M-file functions that you construct to suit your specific application needs.

You execute a callback when a particular event occurs by specifying the name of the M-file callback function as the value for the associated callback property.

This example uses the M-file callback function instrcallback to display a message to the command line when a bytes-available event occurs. The event is generated when the EOSCharCode property value is read.

```
g = gpib('ni',0,1);
g.BytesAvailableFcnMode = 'eosCharCode';
g.BytesAvailableFcn = @instrcallback;
fopen(g)
printf(g,'*IDN?')
readasync(g)
```

The resulting display from instrcallback is shown below.
BytesAvailable event occurred at 17:30:11 for the object: GPIB0-1.

End the GPIB session.

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

You can use the `type` command to display `instrcallback` at the command line.

**Event Types and Callback Properties**
The GPIB event types and associated callback properties are described below.

### GPIB Event Types and Callback Properties

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Associated Property Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes-available</td>
<td>BytesAvailableFcn</td>
</tr>
<tr>
<td></td>
<td>BytesAvailableFcnCount</td>
</tr>
<tr>
<td></td>
<td>BytesAvailableFcnMode</td>
</tr>
<tr>
<td>Error</td>
<td>ErrorFcn</td>
</tr>
<tr>
<td>Output-empty</td>
<td>OutputEmptyFcn</td>
</tr>
<tr>
<td>Timer</td>
<td>TimerFcn</td>
</tr>
<tr>
<td></td>
<td>TimerPeriod</td>
</tr>
</tbody>
</table>

**Bytes-Available Event**

A bytes-available event is generated immediately after a predetermined number of bytes are available in the input buffer or the End-Of-String character 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 `eosCharCode`, then the callback
function executes every time the character specified by the EOSCharCode property is read.

This event can be generated only during an asynchronous read operation.

**Error Event**

An error event is generated immediately after an error, such as 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.

This event executes the callback function specified for the ErrorFcn property. It can be generated only during an asynchronous read or write operation.

**Output-Empty Event**

An output-empty event is generated immediately after the output buffer is empty.

This event executes the callback function specified for the OutputEmptyFcn property. It can be generated only during an asynchronous write operation.

**Timer Event**

A timer event is generated when the time specified by the TimerPeriod property passes. Time is measured relative to when the object is connected to the instrument.

This event executes the callback function specified for the TimerFcn property. Note that some timer events might not be processed if your system is significantly slowed or if the TimerPeriod value is too small.

**Storing Event Information**

You can store 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. As described in “Creating and Executing Callback Functions” on page 4-32, 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 12-6 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.

**GPIB Event Information**

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Field</th>
<th>Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes available</td>
<td>Type</td>
<td>BytesAvailable</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>Error</td>
<td>Type</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td></td>
<td>Data.Message</td>
<td>An error string</td>
</tr>
<tr>
<td>Output empty</td>
<td>Type</td>
<td>OutputEmpty</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>Timer</td>
<td>Type</td>
<td>Timer</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
</tbody>
</table>

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 clock format:

day-month-year hour:minute:second

**Message Field**

Message is used by the error event to store the descriptive message that is generated when an error occurs.
Creating and Executing Callback Functions

You specify the callback function to be executed when a specific event type occurs by including the name of the M-file as the value for the associated callback property. You can specify the callback function as a function handle or as a string cell array element. Function handles are described in the MATLAB function_handle reference page. Note that if you are executing a local callback function from within an M-file, then you must specify the callback as a function handle.

For example, to execute the callback function mycallback every time the EOSCharCode property value is read from your instrument,

    g.BytesAvailableFcnMode = 'eosCharCode';
    g.BytesAvailableFcn = @mycallback;

Alternatively, you can specify the callback function as a cell array.

    g.BytesAvailableFcn = {'mycallback'};

M-file callback functions require at least two input arguments. The first argument is the instrument object. The second argument is a variable that captures the event information given in the preceding table, GPIB Event Information on page 4-31. This event information pertains only to the event that caused the callback function to execute. The function header for mycallback is shown below.

    function mycallback(obj,event)

You pass additional parameters to the callback function by including both the callback function and the parameters as elements of a cell array. For example, to pass the MATLAB variable time to mycallback,

    time = datestr(now,0);
    g.BytesAvailableFcnMode = 'eosCharCode';
    g.BytesAvailableFcn = {@mycallback,time};

Alternatively, you can specify mycallback as a string in the cell array.

    g.BytesAvailableFcn = {'mycallback',time};

The corresponding function header is
function mycallback(obj,event,time)

If you pass additional parameters to the callback function, then they must be included in the function header after the two required arguments.

Note You can also specify the callback function as a string. In this case, the callback is evaluated in the MATLAB workspace and no requirements are made on the input arguments of the callback function.

Enabling Callback Functions After They Error
If an error occurs while a callback function is executing, then

- The callback function is automatically disabled.
- A warning is displayed at the command line, indicating that the callback function is disabled.

If you want to enable the same callback function, you can set the callback property to the same value or you can disconnect the object with the fclose function. If you want to use a different callback function, the callback will be enabled when you configure the callback property to the new value.

Example: Using Events and Callbacks to Read Binary Data
This example extends “Example: Reading Binary Data” on page 4-22 by using the M-file callback function instrcallback to display event-related information to the command line when a bytes-available event occurs during a binary read operation:

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.

```plaintext
g.InputBufferSize = 50000;
g.Timeout = 120;
```

Configure `g` to execute the callback function `instrcallback` every time 5000 bytes is stored in the input buffer. Because `instrcallback` requires an instrument object and event information to be passed as input arguments, the callback function is specified as a function handle.

```plaintext
g.BytesAvailableFcnMode = 'byte';
g.BytesAvailableFcnCount = 5000;
g.BytesAvailableFcn = @instrcallback;
```

3 **Connect to the instrument** — Connect `g` to the oscilloscope.

```plaintext
fopen(g)
```

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

```plaintext
fprintf(g,'HARDCOPY:PORT GPIB')
fprintf(g,'HARDCOPY:FORMAT BMP')
fprintf(g,'HARDCOPY START')
```

Initiate the asynchronous read operation, and begin generating events.

```plaintext
readasync(g)
```

`instrcallback` is called every time 5000 bytes is stored in the input buffer. The resulting displays are shown below.

BytesAvailable event occurred at 09:41:42 for the object: GPIB0-1.
BytesAvailable event occurred at 09:41:50 for the object: GPIB0-1.
BytesAvailable event occurred at 09:41:58 for the object: GPIB0-1.
BytesAvailable event occurred at 09:42:06 for the object: GPIB0-1.
BytesAvailable event occurred at 09:42:14 for the object: GPIB0-1.
BytesAvailable event occurred at 09:42:22 for the object: GPIB0-1.
BytesAvailable event occurred at 09:42:30 for the object: GPIB0-1.
Wait until all the data is sent to the input buffer, and then transfer the data to MATLAB as unsigned 8-bit integers.

```matlab
g.TransferStatus
ans =
idle
out = fread(g,g.BytesAvailable,'uint8');
```

**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.

```matlab
fclose(g)
delete(g)
clear g
```
Triggers

In this section...

“Using the trigger Function” on page 4-36
“Example: Executing a Trigger” on page 4-36

Using the trigger Function

You can execute a trigger with the trigger function. This function is equivalent to writing the GET (Group Execute Trigger) GPIB command to the instrument.

trigger instructs all the addressed Listeners to perform some instrument-specific function such as taking a measurement. Refer to your instrument documentation to learn how to use its triggering capabilities.

Example: Executing a Trigger

This example illustrates GPIB triggering using an Agilent 33120A function generator. The output of the function generator is displayed with an oscilloscope so that you can observe the trigger.

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 Connect to the instrument — Connect g to the function generator.

   fopen(g)

3 Write and read data — Configure the function generator to produce a 5000 Hz sine wave, with 6 volts peak-to-peak.

   fprintf(g,'Func:Shape Sin')
   fprintf(g,'Volt 3')
   fprintf(g,'Freq 5000')
Configure the burst of the trigger to display the sine wave for five seconds, configure the function generator to expect the trigger from the GPIB board, and enable the burst mode.

\begin{verbatim}
fprintf(g,'BM:NCycles 25000')
fprintf(g,'Trigger:Source Bus')
fprintf(g,'BM:State On')
\end{verbatim}

Trigger the instrument.

\begin{verbatim}
trigger(g)
\end{verbatim}

Disable the burst mode.

\begin{verbatim}
fprintf(g,'BM:State Off')
\end{verbatim}

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 below.
4 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.

```matlab
fclose(g)
delete(g)
clear g
```
Serial Polls

Using the spoll Function
You can execute a serial poll with the spoll function. In a serial poll, 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 4-10 for more information on the GPIB bus lines and the RQS bit.

Example: Executing a Serial Poll
This example shows you how to execute a serial poll for an Agilent 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 4-13:

1  Create instrument objects — Create a GPIB object associated with an Agilent 33120A function generator at primary address 1.

        g1 = gpib('ni',0,1);
Create a GPIB object associated with a Tektronix TDS 210 oscilloscope at primary address 2.

\[
g2 = \\text{gpib('ni',0,2)};
\]

2 **Connect to the instrument** — Connect \( g1 \) to the function generator and connect \( g2 \) to the oscilloscope.

\[
\text{fopen([g1 g2])}
\]

3 **Configure property values** — Configure both objects to time out after 1 second.

\[
\text{set([g1 g2],'Timeout',1)}
\]

4 **Write and read data** — Configure the function generator to request service when a command error occurs.

\[
\text{fprintf(g1,'*CLS');} \\
\text{fprintf(g1,'*ESE 32');} \\
\text{fprintf(g1,'*SRE 32');}
\]

Configure the oscilloscope to request service when a command error occurs.

\[
\text{fprintf(g2,'*CLS')}
\text{fprintf(g2,'*PSC 0')}
\text{fprintf(g2,'*ESE 32')}
\text{fprintf(g2,'DESE 32')}
\text{fprintf(g2,'*SRE 32')}
\]

Determine if any instrument needs servicing.

\[
\text{spoll([g1 g2])} \\
\text{ans =} \\
[ ]
\]

Query the voltage value for each instrument.

\[
\text{fprintf(g1,'Volt?')} \\
\text{fprintf(g2,'Volt?')}
\]
Determine if either instrument produced an error due to the preceding query.

```
out = spoll([g1 g2]);
```

Because `Volt?` is an invalid command for the oscilloscope, it is requesting service.

```
out == [g1 g2]
an =
0 1
```

Because `Volt?` is a valid command for the function generator, the value is read back successfully.

```
volt1 = fscanf(g1)
volt1 =
+1.00000E-01
```

However, the oscilloscope read operation times out after 1 second.

```
volt2 = fscanf(g2)
Warning: GPIB: NI: An I/O operation has been canceled, most likely due to a timeout.
```

5 **Disconnect and clean up** — When you no longer need `g1` and `g2`, you should disconnect them from the instruments, and remove them from memory and from the MATLAB workspace.

```
fclose([g1 g2])
delete([g1 g2])
clear g1 g2
```
## Controlling Instruments Using VISA

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VISA Overview (p. 5-3)</td>
<td>Brief description of the Virtual Instrument Standard Architecture (VISA) standard.</td>
</tr>
<tr>
<td>Working with the GPIB Interface (p. 5-5)</td>
<td>The VISA-GPIB object establishes a connection between MATLAB and the instrument via its GPIB interface.</td>
</tr>
<tr>
<td>Working with the VXI Interface (p. 5-9)</td>
<td>The VISA-VXI object establishes a connection between MATLAB and the instrument via its VXI interface.</td>
</tr>
<tr>
<td>Working with the GPIB-VXI Interface (p. 5-22)</td>
<td>The VISA-GPIB-VXI object establishes a connection between MATLAB and the instrument via its GPIB-VXI interface.</td>
</tr>
<tr>
<td>Working with the Serial Port Interface (p. 5-27)</td>
<td>The VISA serial object establishes a connection between MATLAB and the instrument via the serial port.</td>
</tr>
<tr>
<td>Working with the USB Interface (p. 5-31)</td>
<td>The VISA-USB object establishes a connection between MATLAB and the instrument via its USB interface.</td>
</tr>
<tr>
<td>Working with the TCP/IP Interface</td>
<td>The VISA-TCPIP object establishes a connection between MATLAB and the instrument using the instrument’s IP configuration.</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>(p. 5-35)</td>
<td></td>
</tr>
<tr>
<td>Working with the RSIB Interface</td>
<td>The VISA-RSIB object establishes a connection between MATLAB and the instrument via the Rohde &amp; Schwarz VISA passport.</td>
</tr>
<tr>
<td>(p. 5-39)</td>
<td></td>
</tr>
</tbody>
</table>
VISA Overview

In this section...

“What Is VISA?” on page 5-3
“Interfaces Used with VISA” on page 5-3

What Is VISA?

Virtual Instrument Standard Architecture (VISA) is a standard defined by Agilent Technologies and National Instruments for communicating with instruments regardless of the interface.

Instrument Control Toolbox supports the GPIB, VXI, GPIB-VXI, and serial port 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.

Note Most features associated with VISA instrument objects are identical to the features associated with GPIB and serial port objects. Therefore, this chapter presents only interface-specific functions and properties. For example, register-based communication is discussed for VISA-VXI objects, but message-based communication is not discussed as this topic is covered elsewhere in this guide.

Interfaces Used with VISA

For many VISA applications, you can communicate with your instrument without detailed knowledge of how the interface works. In this case, you might want to begin with one of these topics:

- “Working with the GPIB Interface” on page 5-5
- “Working with the VXI Interface” on page 5-9
- “Working with the GPIB-VXI Interface” on page 5-22
- “Working with the Serial Port Interface” on page 5-27
• “Working with the USB Interface” on page 5-31
• “Working with the TCP/IP Interface” on page 5-35
• “Working with the RSIB Interface” on page 5-39

If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to Chapter 2, “The Instrument Control Session”.

**Caution** Instrument Control Toolbox does not support secondary VISA installations. The vendor that you specify when creating a VISA object should refer to the primary VISA installation only. Referring to a secondary VISA installation could generate unpredictable results.
Working with the GPIB Interface

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Understanding VISA-GPIB” on page 5-5</td>
</tr>
<tr>
<td>“Creating a VISA-GPIB Object” on page 5-5</td>
</tr>
<tr>
<td>“The VISA-GPIB Address” on page 5-7</td>
</tr>
</tbody>
</table>

Understanding VISA-GPIB

The GPIB interface is supported through a VISA-GPIB object. The features associated with a VISA-GPIB object are similar to the features associated with a GPIB object. Therefore, only functions and properties that are unique to VISA’s GPIB interface are discussed in this section.

Refer to Chapter 4, “Controlling Instruments Using GPIB”, to learn about the GPIB interface, writing and reading text and binary data, using events and callbacks, using triggers, and so on.

Note The VISA-GPIB object does not support the spoll function, or the BusManagementStatus, CompareBits, and HandshakeStatus properties.

Creating a VISA-GPIB Object

You create a VISA-GPIB object with the visa function. Each VISA-GPIB object is associated with

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

visa requires the vendor name and the resource name as input arguments. The vendor name can be agilent, ni, or tek. 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 for a given instrument with the configuration tool provided by your vendor, or with the instrhwinfo function. (In place of the resource name, you can use an alias as defined with your VISA vendor configuration tool.) As described in
Controlling Instruments Using VISA

“Connecting to the Instrument” on page 2-4, you can also configure properties during object creation.

For example, to create a VISA-GPIB object associated with a National Instruments controller with board index 0, and a Tektronix TDS 210 digital oscilloscope with primary address 1 and secondary address 0,

\[
vg = \text{visa}('ni', 'GPIB0::1::0::INSTR');
\]

The VISA-GPIB object \(vg\) now exists in the MATLAB workspace. You can display the class of \(vg\) with the \texttt{whos} command.

\[
\text{whos } vg
\]

\begin{verbatim}
Name       Size      Bytes  Class
vg         1x1        884  visa object
\end{verbatim}

Grand total is 16 elements using 884 bytes

After you create the VISA-GPIB object, the following properties are automatically assigned values. These properties provide information about the object based on its class type and address information.

**VISA-GPIB Descriptive Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the VISA-GPIB object.</td>
</tr>
<tr>
<td>RsrcName</td>
<td>Indicate the resource name for a VISA instrument.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for \(vg\) with the \texttt{get} function.

\[
\text{get}(vg, \{'\text{Name}','\text{RsrcName}','\text{Type}'\})
\]

\[
\text{ans} = \text{'}VISA-GPIBO-1' \text{'}GPIBO::1::0::INSTR' \text{'}visa-gpib'\]

**VISA-GPIB Object Display**

The VISA-GPIB object provides a convenient display that summarizes important address and state information. You can invoke the display summary as follows:

- Type the VISA-GPIB object at the command line.
- Exclude the semicolon when creating a VISA-GPIB object.
- Exclude the semicolon when configuring properties using 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.

The display summary for the VISA-GPIB object vg is given below.

```
VISA-GPIB Object Using NI Adaptor : VISA-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
```

**The VISA-GPIB Address**

The VISA-GPIB address consists of

- The board index of the GPIB controller installed in your computer.
- The primary address and secondary address of the instrument. Valid primary addresses range from 0 to 30. Valid secondary addresses range...
from 0 to 30, where the value 0 indicates that the secondary address is not used.

You must specify the primary address value via the resource name during VISA-GPIB object creation. Additionally, you must include the board index and secondary address values as part of the resource name if they differ from the default value of 0.

The properties associated with the GPIB address are given below.

**VISA-GPIB Address Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoardIndex</td>
<td>Specify the index number of the GPIB board.</td>
</tr>
<tr>
<td>PrimaryAddress</td>
<td>Specify the primary address of the GPIB instrument.</td>
</tr>
<tr>
<td>SecondaryAddress</td>
<td>Specify the secondary address of the GPIB instrument.</td>
</tr>
</tbody>
</table>

The BoardIndex, PrimaryAddress, and SecondaryAddress properties are automatically updated with the specified resource name values when you create the VISA-GPIB object.

You can display the address property values for the VISA-GPIB object vg created in “Creating a VISA-GPIB Object” on page 5-5 with the get function.

```matlab
get(vg,{'BoardIndex','PrimaryAddress','SecondaryAddress'})
ans =
   [0]    [1]    [0]
```
Working with the VXI Interface

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Understanding VISA-VXI” on page 5-9</td>
</tr>
<tr>
<td>“Creating a VISA-VXI Object” on page 5-10</td>
</tr>
<tr>
<td>“The VISA-VXI Address” on page 5-12</td>
</tr>
<tr>
<td>“Register-Based Communication” on page 5-12</td>
</tr>
</tbody>
</table>

Understanding VISA-VXI

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.
Refer to Chapter 4, “Controlling Instruments Using GPIB”, to learn about general toolbox capabilities such as writing and reading text and binary data, using events and callbacks, and so on.

Creating a VISA-VXI Object

You create a VISA-VXI object with the visa 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

visa requires the vendor name and the resource name as input arguments. The vendor name is either agilent or ni. The resource name consists of the VXI chassis index and the instrument logical address. You can find the VISA-VXI resource name for a given instrument with the configuration tool provided by your vendor, or with the instrhwinfo function. (In place of the resource name, you can use an alias as defined with your VISA vendor configuration tool.) As described in “Connecting to the Instrument” on page 2-4, you can also configure property values during object creation.

For example, to create a VISA-VXI object associated with a VXI chassis with index 0 and an Agilent E1432A 16-channel digitizer with logical address 32,

\[
vv = \text{visa}('agilent', 'VXI0::32::INSTR');
\]

The VISA-VXI object vv now exists in the MATLAB workspace. You can display the class of vv with the whos command.

\[
\begin{array}{llll}
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} \\
vv & 1x1 & 882 & \text{visa object} \\
\end{array}
\]

Grand total is 15 elements using 882 bytes

After you create the VISA-VXI object, the following properties are automatically assigned values. These properties provide information about the object based on its class type and address information.
VISA-VXI Descriptive Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the VISA-VXI object.</td>
</tr>
<tr>
<td>RsrcName</td>
<td>Indicate the resource name for a VISA instrument.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for vv with the get function.

```matlab
get(vv, {'Name', 'RsrcName', 'Type'})
ans =
    'VISA-VXI0-32'    'VXI0::32::INSTR'    'visa-vxi'
```

The VISA-VXI Object Display

The VISA-VXI object provides you with a convenient display that summarizes important address and state information. You can invoke the display summary these three ways:

- Type the VISA-VXI object at the command line.
- Exclude the semicolon when creating a VISA-VXI object.
- Exclude the semicolon when configuring properties using the dot notation.

The display summary for the VISA-VXI object vv is given below.

VISA-VXI Object Using AGILENT Adaptor : VISA-VXI0-32

Communication Address

<table>
<thead>
<tr>
<th>ChassisIndex:</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogicalAddress:</td>
<td>32</td>
</tr>
</tbody>
</table>

Communication State

<table>
<thead>
<tr>
<th>Status:</th>
<th>closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordStatus:</td>
<td>off</td>
</tr>
</tbody>
</table>

Read/Write State

<table>
<thead>
<tr>
<th>TransferStatus:</th>
<th>idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>BytesAvailable:</td>
<td>0</td>
</tr>
</tbody>
</table>
ValuesReceived: 0
ValuesSent: 0

The VISA-VXI Address

The VISA-VXI address consists of

- The chassis index of the VXI chassis
- The logical address of the instrument installed in the VXI chassis

You must specify the logical address value via the resource name during VISA-VXI object creation. Additionally, you must include the chassis index value as part of the resource name if it differs from the default value of 0. The properties associated with the chassis and instrument address are as follows:

**VISA-VXI Address Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChassisIndex</td>
<td>Indicate the index number of the VXI chassis.</td>
</tr>
<tr>
<td>LogicalAddress</td>
<td>Specify the logical address of the VXI instrument.</td>
</tr>
<tr>
<td>Slot</td>
<td>Indicate the slot location of the VXI instrument.</td>
</tr>
</tbody>
</table>

The ChassisIndex and LogicalAddress properties are automatically updated with the specified resource name values when you create the VISA-VXI object. The Slot property is automatically updated after the object is connected to the instrument with the fopen function.

You can display the address property values for the VISA-VXI object vv created in “Creating a VISA-VXI Object” on page 5-10 with the get function.

```matlab
fopen(vv)
get(vv,{'ChassisIndex','LogicalAddress','Slot'})
an =
    [0]    [32]    [2]
```

Register-Based Communication

VXI instruments are either message-based or register-based. Generally, it is assumed that message-based instruments are easier to use, while
register-based instruments are faster. A message-based instrument has its own processor that allows it to interpret high-level commands such as a SCPI command. Therefore, to communicate with a message-based instrument, you can use the read and write functions fscanf, fread, fprintf, and fwrite. For detailed information about these functions, refer to “Communicating with Your Instrument” on page 2-7.

If the message-based instrument also contains shared memory, then you can access the shared memory through register-based read and write operations. A register-based instrument usually does not have its own processor to interpret high-level commands. Therefore, to communicate with a register-based instrument, you need to use read and write functions that access the register.

There are two types of register-based write and read functions: low-level and high-level. The main advantage of the high-level functions is ease of use. Refer to “Example: Using High-Level Memory Functions” on page 5-16 for more information. The main advantage of the low-level functions is speed. Refer to “Example: Using Low-Level Memory Functions” on page 5-19 for more information.

The functions associated with register-based write and read operations are as follows:
VISA-VXI Register-Based Write and Read Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memmap</td>
<td>Map memory for low-level memory read and write operations.</td>
</tr>
<tr>
<td>mempeek</td>
<td>Low-level memory read from VXI register.</td>
</tr>
<tr>
<td>mempoke</td>
<td>Low-level memory write to VXI register.</td>
</tr>
<tr>
<td>memread</td>
<td>High-level memory read from VXI register.</td>
</tr>
<tr>
<td>memunmap</td>
<td>Unmap memory for low-level memory read and write operations.</td>
</tr>
<tr>
<td>memwrite</td>
<td>High-level memory write to VXI register.</td>
</tr>
</tbody>
</table>

The properties associated with register-based write and read operations are given below.
VISA-VXI Register-Based Write and Read Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MappedMemoryBase</td>
<td>Indicate the base memory address of the mapped memory.</td>
</tr>
<tr>
<td>MappedMemorySize</td>
<td>Indicate the size of the mapped memory for low-level read and write operations.</td>
</tr>
<tr>
<td>MemoryBase</td>
<td>Indicate the base address of the A24 or A32 space.</td>
</tr>
<tr>
<td>MemoryIncrement</td>
<td>Specify if the VXI register offset increments after data is transferred.</td>
</tr>
<tr>
<td>MemorySize</td>
<td>Indicate the size of the memory requested in the A24 or A32 address space.</td>
</tr>
<tr>
<td>MemorySpace</td>
<td>Define the address space used by the instrument.</td>
</tr>
</tbody>
</table>

Example: Understanding Your Instrument’s Register Characteristics

This example explores the register characteristics for an Agilent E1432A 16-channel 51.2 kSa/s digitizer with a DSP module.

All VXI instruments have an A16 memory space consisting of 64 bytes. It is known as an A16 space because the addresses are 16 bits wide. Register-based instruments provide a memory map of the address space that describes the information contained within the A16 space. Some VXI instruments also have an A24 or A32 space if the 64 bytes provided by the A16 space are not enough to perform the necessary tasks. A VXI instrument cannot use both the A24 and A32 space:

1 Create an instrument object — Create the VISA-VXI object vv associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

   \[
   \text{vv} = \text{visa('agilent','VXI0::130::INSTR')};
   \]

2 Connect to the instrument — Connect vv to the instrument.

   \[
   \text{fopen(vv)}
   \]
The `MemorySpace` property indicates the type of memory space the instrument supports. By default, all instruments support A16 memory space. However, this property can be A16/A24 or A16/A32 if the instrument also supports A24 or A32 memory space, respectively.

```matlab
get(vv,'MemorySpace')
an =
A16/A24
```

If the VISA-VXI object is not connected to the instrument, `MemorySpace` always returns the default value of A16.

The `MemoryBase` property indicates the base address of the A24 or A32 space, and is defined as a hexadecimal string. The `MemorySize` property indicates the size of the A24 or A32 space. If the VXI instrument supports only the A16 memory space, `MemoryBase` defaults to 0H and `MemorySize` defaults to 0.

```matlab
get(vv,{'MemoryBase','MemorySize'})
an =
'200000H' [262144]
```

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

```matlab
fclose(vv)
delete(vv)
clear vv
```

**Example: Using High-Level Memory Functions**

This example uses the high-level memory functions, `memread` and `memwrite`, to access register information for an Agilent E1432A 16-channel 51.2 kSa/s digitizer with a DSP module. The main advantage of these high-level functions is ease of use — you can access multiple registers with one function call, and the memory that is to be accessed is automatically mapped for you. The main disadvantage is the lack of speed — they are slower than the low-level memory functions.
Each register contains 16 bits, and is associated with an offset value that you supply to `memread` or `memwrite`. The first four registers of the digitizer are accessed in this example, and are described below.

**Agilent E1432A Register Information**

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>0</td>
<td>This register provides instrument configuration information and is always defined as CFFF. Bits 15 and 14 are 1, indicating that the instrument is register-based. Bits 13 and 12 are 0, indicating that the instrument supports the A24 memory space. The remaining bits are all 1, indicating the device ID.</td>
</tr>
<tr>
<td>Device Type</td>
<td>2</td>
<td>This register provides instrument configuration information. Bits 15-12 indicate the memory required by the A24 space. The remaining bits indicate the model code for the instrument.</td>
</tr>
<tr>
<td>Status</td>
<td>4</td>
<td>This register provides instrument status information. For example, bit 15 indicates whether you can access the A24 registers, and bit 6 indicates whether a DSP communication error occurred.</td>
</tr>
<tr>
<td>Offset</td>
<td>6</td>
<td>This register defines the base address of the instrument’s A24 registers. Bits 15-12 map the VME Bus address lines A23-A20 for A24 register access. The remaining bits are all 0.</td>
</tr>
</tbody>
</table>

For more detailed information about these registers, refer to the *HP E1432A User’s Guide*.

1 **Create an instrument object** — Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address is 130.

   ```matlab
   vv = visa('agilent','VXI0::130::INSTR');
   ```

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

   ```matlab
   fopen(vv)
   ```
**Write and read data** — The following command performs a high-level read of the ID Register, which has an offset of 0.

```matlab
reg1 = memread(vv,0,'uint16','A16')
reg1 =
    53247
```

Convert `reg1` to a hexadecimal value and a binary string. Note that the hex value is CFFF and the least significant 12 bits are all 1, as expected.

```matlab
dec2hex(reg1)
anS =
    CFFF
dec2bin(reg1)
anS =
    1100111111111111
```

You can read multiple registers with `memread`. The following command reads the next three registers. An offset of 2 indicates that the read operation begins with the Device Type Register.

```matlab
reg24 = memread(vv,2,'uint16','A16',3)
reg24 =
     20993
     50012
     40960
```

The following commands write to the Offset Register and then read the value back. Note that if you change the value of this register, you will not be able to access the A24 space.

```matlab
memwrite(vv,45056,6,'uint16','A16');
reg4 = memread(vv,6,'uint16','A16')
reg4 =
    45056
```

Note that the least significant 12 bits are all 0, as expected.

```matlab
dec2bin(reg4,16)
anS =
    1011000000000000
```
Restore the original register value, which is stored in the `reg24` variable.

```matlab
memwrite(vv,reg24(3),6,'uint16','A16');
```

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

```matlab
fclose(vv)
delete(vv)
clear vv
```

**Example: Using Low-Level Memory Functions**

This example uses the low-level memory functions `mempeek` and `mempoke` to access register information for an Agilent E1432A 16-channel 51.2 kSa/s digitizer with a DSP module. The main advantage of these low-level functions is speed — they are faster than the high-level memory functions. The main disadvantages include the inability to access multiple registers with one function call, errors are not reported, and you must map the memory that is to be accessed.

For information about the digitizer registers accessed in this example, refer to “Example: Using High-Level Memory Functions” on page 5-16:

1 **Create an instrument object** — Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

```matlab
vv = visa('agilent','VXI0::130::INSTR');
```

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

```matlab
fopen(vv)
```

3 **Write and read data** — Before you can use the low-level memory functions, you must first map the memory space with the `memmap` function. If the memory requested by `memmap` does not exist, an error is returned. The following command maps the first 16 registers of the A16 memory.

```matlab
memmap(vv,'A16',0,16);
```
The MappedMemoryBase and MappedMemorySize properties indicate if memory has been mapped. MappedMemoryBase is the base address of the mapped memory and is defined as a hexadecimal string. MappedMemorySize is the size of the mapped memory. These properties are similar to the MemoryBase and MemorySize properties that describe the A24 or A32 memory space.

```matlab
get(vv,{'MappedMemoryBase','MappedMemorySize'})
ans =
    '16737610H'    [16]
```

The following command performs a low-level read of the ID Register, which has an offset of 0.

```matlab
reg1 = mempeek(vv,0,'uint16')
reg1 =
    53247
```

The following command performs a low-level read of the Offset Register, which has an offset of 6.

```matlab
reg4 = mempeek(vv,6,'uint16')
reg4 =
    40960
```

The following commands write to the Offset Register and then read the value back. Note that if you change the value of this register, you will not be able to access the A24 space.

```matlab
mempoke(vv,45056,6,'uint16');
mempoke(vv,6,'uint16')
ans =
    45056
```

Restore the original register value.

```matlab
mempoke(vv,reg4,6,'uint16');
```

When you have finished accessing the registers, you should unmap the memory with the memunmap function.

```matlab
memunmap(vv)
```
get(vv,{'MappedMemoryBase','MappedMemorySize'})
ans =
    '0H' [0]

If memory is still mapped when the object is disconnected from the instrument, the memory is automatically unmapped for you.

4 Disconnect and clean up — When you no longer need vv, you should disconnect it from the instrument, and remove it from memory and from the MATLAB workspace.

fclose(vv)
delete(vv)
clear vv
Working with the GPIB-VXI Interface

In this section...

“Understanding VISA_GPIB_VXI” on page 5-22
“Creating a VISA-GPIB-VXI Object” on page 5-23
“The VISA-GPIB-VXI Address” on page 5-25

Understanding VISA_GPIB_VXI

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

The GPIB-VXI interface is supported through a VISA-GPIB-VXI object. The features associated with a VISA-GPIB-VXI object are similar to the features associated with GPIB and VISA-VXI objects. Therefore, only functions and properties that are unique to VISA's GPIB-VXI interface are discussed in this section.
Refer to Chapter 4, “Controlling Instruments Using GPIB”, to learn about writing and reading text and binary data, using events and callbacks, using triggers, and so on. Refer to “Register-Based Communication” on page 5-12, to learn about accessing VXI registers.

**Note** The VISA-GPIB-VXI object does not support the spoll and trigger functions, or the BusManagementStatus, HandshakeStatus, InterruptFcn, TriggerFcn, TriggerLine, and TriggerType properties.

**Creating a VISA-GPIB-VXI Object**

You create a VISA-GPIB-VXI object with the `visa` function. As shown in the preceding figure, each object is associated with the following:

- A GPIB controller installed in your computer
- A VXI chassis
- A GPIB-VXI command module in slot 0 of the VXI chassis
- An instrument installed in the VXI chassis

`visa` requires the vendor name and the resource name as input arguments. The vendor name is either `agilent` or `ni`. The resource name consists of the VXI chassis index and the instrument logical address. You can find the VISA-GPIB-VXI resource name for a given instrument with the configuration tool provided by your vendor, or with the `instrhwinfo` function. (In place of the resource name, you can use an alias as defined with your VISA vendor configuration tool.) As described in “Connecting to the Instrument” on page 2-4, you can also configure property values during object creation.

For example, to create a VISA-GPIB-VXI object associated with a VXI chassis with index 0, an Agilent E1406A Command Module in slot 0, and an Agilent E1441A Arbitrary Waveform Generator in slot 2 with logical address 80,

```matlab
vgv = visa('agilent','GPIB-VXI0::80::INSTR');
```

The VISA-GPIB-VXI object `vgv` now exists in the MATLAB workspace. You can display the class of `vgv` with the `whos` command.

```matlab
whos vgv
```
Controlling Instruments Using VISA

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>vgv</td>
<td>1x1</td>
<td>892</td>
<td>visa object</td>
</tr>
</tbody>
</table>

Grand total is 20 elements using 892 bytes

After you create the VISA-GPIB-VXI object, the properties listed below are automatically assigned values. These properties provide descriptive information about the object based on its class type and address information.

**VISA-GPIB-VXI Descriptive Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the VISA-GPIB-VXI object.</td>
</tr>
<tr>
<td>RsrcName</td>
<td>Indicate the resource name for a VISA instrument.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for vgv with the `get` function.

```matlab
get(vgv,{'Name','RsrcName','Type'})
```

```matlab
ans =
'VISA-GPIB-VXI-80'    'GPIB-VXI::80::INSTR'    'visa-gpib-vxi'
```

**Note** The GPIB-VXI communication interface is a combination of the GPIB and VXI interfaces. Therefore, you can also use a VISA-GPIB object to communicate with instruments installed in the VXI chassis, or to communicate with non-VXI instruments connected to the slot 0 controller.

**The VISA-GPIB-VXI Object Display**

The VISA-GPIB-VXI object provides you with a convenient display that summarizes important address and state information. You can invoke the display summary these three ways:

- Type the VISA-GPIB-VXI object at the command line.
- Exclude the semicolon when creating a VISA-GPIB-VXI object.
- Exclude the semicolon when configuring properties using the dot notation.

The display summary for the VISA-GPIB-VXI object vgv is given below.

VISA-GPIB-VXI Object Using AGILENT Adaptor : VISA-GPIB-VXIO-80

Communication Address
  ChassisIndex: 0
  LogicalAddress: 80

Communication State
  Status: closed
  RecordStatus: off

Read/Write State
  TransferStatus: idle
  BytesAvailable: 0
  ValuesReceived: 0
  ValuesSent: 0

The VISA-GPIB-VXI Address
The VISA-GPIB-VXI address consists of a VXI component and a GPIB component. The VXI component includes the following:

- The chassis index of the VXI chassis
- The logical address of the VXI instrument; the logical address must be 0, or it must be divisible by 8
- The slot of the VXI instrument

The GPIB component includes

- The board index of the GPIB controller installed in your computer
- The primary address of the GPIB-VXI command module in slot 0
- The secondary address of the VXI instrument
You must specify the logical address value via the resource name during VISA-GPIB-VXI object creation. Additionally, you must include the chassis index value as part of the resource name if it differs from the default value of 0. The properties associated with the VISA-GPIB-VXI address are given below.

**VISA-GPIB-VXI Address Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoardIndex</td>
<td>Indicate the index number of the GPIB board.</td>
</tr>
<tr>
<td>ChassisIndex</td>
<td>Specify the index number of the VXI chassis.</td>
</tr>
<tr>
<td>LogicalAddress</td>
<td>Specify the logical address of the VXI instrument.</td>
</tr>
<tr>
<td>PrimaryAddress</td>
<td>Indicate the primary address of the GPIB-VXI command module.</td>
</tr>
<tr>
<td>SecondaryAddress</td>
<td>Indicate the secondary address of the VXI instrument.</td>
</tr>
<tr>
<td>Slot</td>
<td>Indicate the slot location of the VXI instrument.</td>
</tr>
</tbody>
</table>

The ChassisIndex and LogicalAddress properties are automatically updated with the specified resource name values when you create the VISA-GPIB-VXI object. The BoardIndex, PrimaryAddress, SecondaryAddress, and Slot properties are automatically updated after the object is connected to the instrument with the fopen function.

You can display the address property values for the VISA-GPIB-VXI object vgv created in “Creating a VISA-GPIB-VXI Object” on page 5-23 with the get function.

```matlab
fopen(vgv)
get(vgv,{'BoardIndex','ChassisIndex','LogicalAddress',...
'PrimaryAddress','SecondaryAddress','Slot'})
ans =
    [0]    [0]    [80]    [9]    [10]    [2]
```
Working with the Serial Port Interface

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Understanding the Serial Port” on page 5-27</td>
</tr>
<tr>
<td>“Creating a VISA-Serial Object” on page 5-27</td>
</tr>
<tr>
<td>“Configuring Communication Settings” on page 5-29</td>
</tr>
</tbody>
</table>

Understanding the Serial Port

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 serial port object. Therefore, only functions and properties that are unique to VISA's serial port interface are discussed in this section.

Refer to Chapter 6, “Controlling Instruments Using the Serial Port”, to learn about writing and reading text and binary data, using events and callbacks, using serial port control lines, and so on.

Note The VISA-serial object does not support the serialbreak function, the BreakInterruptFcn property, and the PinStatusFcn property.

Creating a VISA-Serial Object

You create a VISA-serial object with the visa function. Each VISA-serial object is associated with an instrument connected to a serial port on your computer.

visa requires the vendor name and the resource name as input arguments. The vendor name can be agilent, ni, or tek. The resource name consists of the name of the serial port connected to your instrument. You can find the VISA-serial resource name for a given instrument with the configuration tool provided by your vendor, or with the instrhwinfo function. (In place of the resource name, you can use an alias as defined with your VISA vendor configuration tool.) As described in “Connecting to the Instrument” on page 2-4, you can also configure property values during object creation.
For example, to create a VISA-serial object that is associated with the COM1 port, and that uses National Instruments VISA,

```matlab
vs = visa('ni','ASRL1::INSTR');
```

The VISA-serial object `vs` now exists in the MATLAB workspace. You can display the class of `vs` with the `whos` command.

```matlab
whos vs
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs</td>
<td>1x1</td>
<td>888</td>
<td>visa object</td>
</tr>
</tbody>
</table>

Grand total is 18 elements using 888 bytes

After you create the VISA-serial object, the properties listed below are automatically assigned values. These properties provide descriptive information about the object based on its class type and address information.

**VISA-Serial Descriptive Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the VISA-serial object.</td>
</tr>
<tr>
<td>Port</td>
<td>Indicate the serial port name.</td>
</tr>
<tr>
<td>RsrcName</td>
<td>Indicate the resource name for a VISA instrument.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for `vs` with the `get` function.

```matlab
get(vs,{'Name','Port','RsrcName','Type'})
```

```
ans =
'VISA-Serial-ASRL1' 'ASRL1' 'ASRL1::INSTR' 'visa-serial'
```

**The VISA-Serial Object Display**
The VISA-serial object provides you with a convenient display that summarizes important address and state information. You can invoke the display summary these three ways:
• Type the VISA-serial object at the command line.
• Exclude the semicolon when creating a VISA-serial object.
• Exclude the semicolon when configuring properties using the dot notation.

The display summary for the VISA-serial object vs is given below.

VISA-Serial Object Using NI Adaptor : VISA-Serial-ASRL1

Communication Settings
  Port: ASRL1
  BaudRate: 9600
  Terminator: 'LF'

Communication State
  Status: closed
  RecordStatus: off

Read/Write State
  TransferStatus: idle
  BytesAvailable: 0
  ValuesReceived: 0
  ValuesSent: 0

Configuring 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 6-9. These properties are given below.
VISA-Serial Communication Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaudRate</td>
<td>Specify the rate at which bits are transmitted.</td>
</tr>
<tr>
<td>DataBits</td>
<td>Specify the number of data bits to transmit.</td>
</tr>
<tr>
<td>Parity</td>
<td>Specify the type of parity checking.</td>
</tr>
<tr>
<td>StopBits</td>
<td>Specify the number of bits used to indicate the end of a byte.</td>
</tr>
<tr>
<td>Terminator</td>
<td>Specify the character used to terminate commands written to the instrument.</td>
</tr>
</tbody>
</table>

Refer to your instrument documentation for an explanation of its supported communication settings. Note that the valid values for StopBits are 1 and 2 and the valid values for Terminator do not include CR/LF and LF/CR. These property values differ from the values supported for the serial port object.

You can display the default communication property values for the VISA-serial object vs created in “Creating a VISA-Serial Object” on page 5-27 with the get function.

```matlab
get(vs,{'BaudRate','DataBits','Parity','StopBits','Terminator'})
ans =
    [9600]    [8]      'none'    [1]      'LF'
```
Working with the USB Interface

In this section...

“Creating a VISA-USB Object” on page 5-31
“The VISA-USB Address” on page 5-33

Creating a VISA-USB Object

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

`visa` requires the vendor name and the resource name as input arguments. The vendor name can be `agilent`, `ni`, or `tek`. The resource name consists of the USB board index, manufacturer ID, model code, serial number, and interface number of the connected instrument. You can find the VISA-USB resource name for a given instrument with the configuration tool provided by your vendor, or with the `instrhwinfo` function. (In place of the resource name, you can use an alias as defined with your VISA vendor configuration tool.) As described in “Connecting to the Instrument” on page 2-4, you can also configure property values during object creation.

For example, to create a VISA-USB object that uses National Instruments VISA,

```matlab
vu = visa('ni','USB::0x1234::125::A22-5::INSTR');
```

The VISA-USB object `vu` now exists in the MATLAB workspace. You can display the class of `vu` with the `whos` command.

```matlab
whos vu
```

```
Name  Size   Bytes  Class
 vu   1x1     882  visa object
```

Grand total is 15 elements using 882 bytes

After you create the VISA-USB object, the properties listed below are automatically assigned values. These properties provide descriptive information about the object based on its class type and address information.
**VISA-USB Descriptive Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the VISA-USB object.</td>
</tr>
<tr>
<td>RsccName</td>
<td>Indicate the resource name for a VISA instrument.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for `vs` with the `get` function.

```matlab
get(vu,{'Name','RsccName','Type'}))
ans =
'VISA-USB-0-0x1234-125-A22-5-0'    'USB::0x1234::125::A22-5::INSTR'
'visa-usb'
```

**The VISA-USB Object Display**

The VISA-USB object provides you with a convenient display that summarizes important address and state information. You can invoke the display summary these three ways:

- Type the VISA-USB object at the command line.
- Exclude the semicolon when creating a VISA-USB object.
- Exclude the semicolon when configuring properties using the dot notation.

The display summary for the VISA-USB object `vs` is given below.

```
VISA-USB Object Using NI Adaptor : VISA-USB-0-0x1234-125-A22-5-0

Communication Address
   ManufacturerID:    0x1234
   ModelCode:         125
   SerialNumber:      A22-5

Communication State
   Status:            closed
   RecordStatus:      off
```
Read/Write State

- TransferStatus: idle
- BytesAvailable: 0
- ValuesReceived: 0
- ValuesSent: 0

**The VISA-USB Address**

The VISA-USB address consists of

- Board index (optional, from the VISA configuration)
- Manufacturer ID of the instrument
- Model code of the instrument
- Serial number of the instrument
- Interface number (optional, from the VISA configuration)

You specify these address property values via the resource name during VISA-USB object creation. The instrument address properties are given below.

**VISA-USB Address Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoardIndex</td>
<td>Specify the index number of the USB board in VISA configuration (optional — defaults to 0).</td>
</tr>
<tr>
<td>InterfaceIndex</td>
<td>Specify the USB interface number (optional).</td>
</tr>
<tr>
<td>ManufacturerID</td>
<td>Specify the manufacturer ID of the USB instrument.</td>
</tr>
<tr>
<td>ModelCode</td>
<td>Specify the model code of the USB instrument.</td>
</tr>
<tr>
<td>SerialNumber</td>
<td>Specify the index of the USB instrument on the USB hub.</td>
</tr>
</tbody>
</table>

The properties are automatically updated with the specified resource name values when you create the VISA-USB object.

With the `get` function, you can display the address property values for the VISA-USB object `vu`, created in “Creating a VISA-USB Object” on page 5-31.
fopen(vu)
get(vu,{'ManufacturerID','ModelCode','SerialNumber'})
ans =
    [0x1234]   [125]   [A22-5]
Working with the TCP/IP Interface

In this section...

“Understanding VISA-TCP/IP” on page 5-35
“Creating a VISA-TCPIP Object” on page 5-35
“The VISA-TCPIP Address” on page 5-37

Understanding VISA-TCP/IP

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 tcpip object. Therefore, only functions and properties that are unique to VISA’s TCP/IP interface are discussed in this section.

Refer to Chapter 7, “Controlling Instruments Using TCP/IP and UDP”, to learn about writing and reading text and binary data, using events and callbacks, and so on.

Creating a VISA-TCPIP Object

You create a VISA-TCPIP object with the visa function. Each VISA-TCPIP object is associated with an instrument connected to your computer.

visa requires the vendor name and the resource name as input arguments. The vendor name can be agilent, ni, or tek. The resource name consists of the TPC/IP board index, IP address or host name, and LAN device name of your instrument. You can find the VISA-TCPIP resource name for a given instrument with the configuration tool provided by your vendor, or with the instrhwinfo function. (In place of the resource name, you can use an alias as defined with your VISA vendor configuration tool.) As described in “Connecting to the Instrument” on page 2-4, you can also configure property values during object creation.

For example, to create a VISA-TCPIP object that uses National Instruments VISA associated with an instrument at IP address 216.148.60.170,

\[
vt = \text{visa}('ni', 'TCPIP::216.148.60.170::INSTR');
\]
The VISA-TCPIP object vs now exists in the MATLAB workspace. You can display the class of vs with the whos command.

```matlab
whos vt
Name       Size    Bytes  Class
vt         1x1      886    visa object
```

Grand total is 17 elements using 886 bytes

After you create the VISA-TCPIP object, the properties listed below are automatically assigned values. These properties provide descriptive information about the object based on its class type and address information.

**VISA-TCPIP Descriptive Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the VISA-TCPIP object.</td>
</tr>
<tr>
<td>RsrcName</td>
<td>Indicate the resource name for a VISA instrument.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for vs with the get function.

```matlab
get(vt,{'Name','RsrcName','Type'})
an Wilmington 6.148.60.170'  'TCPIP::216.148.60.170::INSTR' ' visa-tcpip'
```

**The VISA-TCPIP Object Display**

The VISA-TCPIP object provides you with a convenient display that summarizes important address and state information. You can invoke the display summary these three ways:

- Type the VISA-TCPIP object at the command line.
- Exclude the semicolon when creating a VISA-TCPIP object.
- Exclude the semicolon when configuring properties using the dot notation.
The display summary for the VISA-TCPIP object vs is given below.

VISA-TCPIP Object Using NI Adaptor : VISA-TCPIP-216.148.60.170

Communication Address
  RemoteHost: 216.148.60.170

Communication State
  Status: closed
  RecordStatus: off

Read/Write State
  TransferStatus: idle
  BytesAvailable: 0
  ValuesReceived: 0
  ValuesSent: 0

The VISA-TCPIP Address
The VISA-TCPIP address consists of

- Board index (optional, from the VISA configuration)
- Remote host of the instrument
- LAN device name of the instrument (optional)

You specify these address property values via the resource name during VISA-TCPIP object creation. The instrument address properties are given below.

VISA-TCPIP Address Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoardIndex</td>
<td>Specify the index number of the TCP/IP board in VISA configuration (optional — defaults to 0).</td>
</tr>
</tbody>
</table>
**VISA-TCPIP Address Properties (Continued)**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemoteHost</td>
<td>Specify the remote host name or IP address of the instrument.</td>
</tr>
<tr>
<td>LANName</td>
<td>Specify the LAN device name of the instrument.</td>
</tr>
</tbody>
</table>

The properties are automatically updated with the specified resource name values when you create the VISA-TCPIP object.

With the `get` function, you can display the address property values for the VISA-TCPIP object `vt`, created in “Creating a VISA-TCPIP Object” on page 5-35.

```matlab
fopen(vt)
get(vt,{'RemoteHost'})
an =
    [216.148.60.170]
```
Creating a VISA-RSIB Object

You create a VISA-RSIB object with the `visa` function. Each VISA-RSIB object is associated with an instrument connected to your computer.

`visa` requires the vendor name and the resource name as input arguments. The only supported vendor name is `ni`. The resource name consists of the IP address or host name of the instrument. You can find the VISA-RSIB resource name for a given instrument with the configuration tool provided by your vendor, or with the `instrhwinfo` function. (In place of the resource name, you can use an alias as defined with your VISA vendor configuration tool.) As described in “Connecting to the Instrument” on page 2-4, you can also configure properties during object creation.

For example, to create a VISA-RSIB object that uses National Instruments VISA and associated with an instrument with IP address 192.168.1.33,

```
vr = visa('ni','RSIB::192.168.1.33::INSTR');
```

The VISA-RSIB object `vr` now exists in the MATLAB workspace. You can display the class of `vr` with the `whos` command.

```
whos vr
Name        Size    Bytes  Class
vr           1x1      884   visa object

Grand total is 16 elements using 884 bytes
```

After you create the VISA-RSIB object, the properties listed below are automatically assigned values. These properties provide descriptive information about the object based on its class type and address information.
VISA-RSIB Descriptive Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the VISA-RSIB object.</td>
</tr>
<tr>
<td>RsrvName</td>
<td>Indicate the resource name for a VISA instrument.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for `vr` with the `get` function.

```matlab
get(vr, {'Name', 'RsrcName', 'Type'})
```

```
ans =
    'VISA-RSIB0-192.168.1.33'    'RSIB0::192.168.1.33::INSTR'
    'visa-RSIB'
```

The VISA-RSIB Object Display

The VISA-RSIB object provides you with a convenient display that summarizes important address and state information. You can invoke the display summary these three ways:

- Type the VISA-RSIB object at the command line.
- Exclude the semicolon when creating a VISA-RSIB object.
- Exclude the semicolon when configuring properties 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.

The display summary for the VISA-RSIB object `vr` is given below.

VISA-RSIB Object Using NI Adaptor : VISA-RSIB-192.168.1.33

Communication Address
RemoteHost: 192.168.1.33

Communication State
Status: closed
RecordStatus: off
Read/Write State
  TransferStatus:  idle
  BytesAvailable:  0
  ValuesReceived:  0
  ValuesSent:  0

The VISA-RSIB Address
The VISA-RSIB address consists of

- Remote host of the instrument

You specify the address property value via the resource name during VISA-RSIB object creation. The instrument address property is given below.

### VISA-RSIB Address Property

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemoteHost</td>
<td>Specify the remote host name or IP address of the instrument</td>
</tr>
</tbody>
</table>

The property is automatically updated with the specified resource name value when you create the VISA-RSIB object.

With the `get` function, you can display the address property value for the VISA-RSIB object `vr`, created in “Creating a VISA-RSIB Object” on page 5-39.

```matlab
fopen(vr)
get(vr,{'RemoteHost'})
an =
    [192.168.1.33]
```
Controlling Instruments Using the Serial Port

This chapter describes specific issues related to controlling instruments that use the serial port. The sections are as follows.

Serial Port Overview (p. 6-2) Basic features of the serial port.
The Serial Port Object (p. 6-16) The serial port object establishes a connection between MATLAB and the instrument via serial port.
Configuring Communication Settings (p. 6-19) Communication settings are associated with the baud rate and serial data format.
Writing and Reading Data (p. 6-20) Port-specific issues related to writing and reading data with a serial port object.
Events and Callbacks (p. 6-25) Enhance your instrument control application using events and callbacks.
Using Control Pins (p. 6-30) The control pins allow you to signal the presence of connected devices and to control the flow of data.
Serial Port Overview

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“What Is Serial Communication?” on page 6-2</td>
</tr>
<tr>
<td>“The Serial Port Interface Standard” on page 6-3</td>
</tr>
<tr>
<td>“Supported Platforms” on page 6-3</td>
</tr>
<tr>
<td>“Connecting Two Devices with a Serial Cable” on page 6-3</td>
</tr>
<tr>
<td>“Serial Port Signals and Pin Assignments” on page 6-5</td>
</tr>
<tr>
<td>“Serial Data Format” on page 6-9</td>
</tr>
<tr>
<td>“Finding Serial Port Information for Your Platform” on page 6-13</td>
</tr>
</tbody>
</table>

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 “The Serial Port Object” on page 6-16. If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to Chapter 2, “The Instrument Control Session”.
The 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 serial port object. Of these, the most widely used standard is RS-232, which stands for Recommended Standard number 232.

The current version of this standard is designated as TIA/EIA-232C, which is published by the Telecommunications Industry Association. However, the term “RS-23” is still in popular use, and is used in this guide when referring to a serial communication port that follows the TIA/EIA-232 standard. RS-232 defines these serial port characteristics:

- The maximum bit transfer rate and cable length
- The names, electrical characteristics, and functions of signals
- The mechanical connections and pin assignments

Primary communication is accomplished using 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 In this guide, it is assumed 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 Microsoft Windows 32-bit, Linux 32-bit, and Sun Solaris 64-bit platforms.

Connecting Two Devices with a Serial Cable

The RS-232 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.
Throughout 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 assignments are defined such that straight-through cabling is used, 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 6-5 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 serial port 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 pin assignment scheme for a 9-pin male connector on a DTE is given below.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Label</th>
<th>Signal Name</th>
<th>Signal Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CD</td>
<td>Carrier Detect</td>
<td>Control</td>
</tr>
<tr>
<td>2</td>
<td>RD</td>
<td>Received Data</td>
<td>Data</td>
</tr>
<tr>
<td>3</td>
<td>TD</td>
<td>Transmitted Data</td>
<td>Data</td>
</tr>
<tr>
<td>4</td>
<td>DTR</td>
<td>Data Terminal Ready</td>
<td>Control</td>
</tr>
<tr>
<td>5</td>
<td>GND</td>
<td>Signal Ground</td>
<td>Ground</td>
</tr>
<tr>
<td>6</td>
<td>DSR</td>
<td>Data Set Ready</td>
<td>Control</td>
</tr>
<tr>
<td>7</td>
<td>RTS</td>
<td>Request to Send</td>
<td>Control</td>
</tr>
<tr>
<td>8</td>
<td>CTS</td>
<td>Clear to Send</td>
<td>Control</td>
</tr>
<tr>
<td>9</td>
<td>RI</td>
<td>Ring Indicator</td>
<td>Control</td>
</tr>
</tbody>
</table>

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 “on” and “off” states for a data signal and for a control signal are shown below.
Serial Port Overview

The 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. In this guide, it is assumed 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.

The Control Pins
The control pins of a 9-pin serial port are used to determine the presence of connected devices and control the flow of data. The control pins include

- “The RTS and CTS Pins” on page 6-8
- “The DTR and DSR Pins” on page 6-8
- “The CD and RI Pins” on page 6-8
**The 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 6-33.

**The 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. However, you should refer to your device documentation to determine its specific pin behavior.

**The 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’s 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. The diagram below illustrates the serial data format.

![Serial Data Format Diagram]

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 “Writing and Reading Data” on page 6-20.

**Synchronous and Asynchronous Communication**

The RS-232 standard supports 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** When used in this guide, the terms “synchronous” and “asynchronous” refer to whether read or write operations block access to the MATLAB command line.
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 6-10, 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(s) 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 might 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.

**The Parity Bit**

The parity bit provides simple error (parity) checking for the transmitted data. The types of parity checking are given below.

**Parity Types**

<table>
<thead>
<tr>
<th>Parity Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Even</td>
<td>The data bits plus the parity bit produce an even number of 1s.</td>
</tr>
<tr>
<td>Mark</td>
<td>The parity bit is always 1.</td>
</tr>
<tr>
<td>Odd</td>
<td>The data bits plus the parity bit produce an odd number of 1s.</td>
</tr>
<tr>
<td>Space</td>
<td>The parity bit is always 0.</td>
</tr>
</tbody>
</table>

Mark and space parity checking are seldom used because they offer minimal error detection. You might 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 1 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.

Finding Serial Port Information for Your Platform
This section describes how to 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 will have no effect on your serial port application.

You can also use the instrhwinfo function to return the available serial ports programmatically.

Windows Platform
You can access serial port information through the System Properties dialog box. To access this in Window XP,

1 Right-click My Computer on the desktop, and select Properties.
2 In the System Properties dialog box, click the Hardware tab.
3 Click Device Manager.
4 In the Device Manager dialog box, expand the Ports node.
5 Double-click the Communications Port (COM1) node.
6 Select the Port Settings tab.
The resulting **Ports** dialog box is shown below.

![Ports dialog box](image)

**UNIX Platform**

To find serial port information for UNIX platforms, you need to know the serial port names. These names might 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,

```bash
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`,

```bash
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: infinite
  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, you can use the `stty` command to display or configure serial port information. For example, to display serial port properties for `ttyS0`,

```
stty -a < /dev/ttyS0
```

To configure the baud rate to 4800 bits per second,

```
stty speed 4800 < /dev/ttyS0 > /dev/ttyS0
```
The Serial Port Object

In this section...

“Creating a Serial Port Object” on page 6-16
“The Serial Port Object Display” on page 6-17

Creating a Serial Port Object

You create a serial port object with the `serial` function. `serial` requires the name of the serial port connected to your device as an input argument. As described in “Configuring Properties During Object Creation” on page 3-3, you can also configure property values during object creation.

Each serial port object is associated with one serial port. For example, to create a serial port object associated with the COM1 port,

```matlab
s = serial('COM1');
```

The serial port object `s` now exists in the MATLAB workspace. You can display the class of `s` with the `whos` command.

```matlab
whos s
Name      Size  Bytes  Class          Value
----------  -----  ------  -------------  ----
s          1x1    512 serial object
```

Once the serial port object is created, the following properties are automatically assigned values. These general purpose properties provide information about the serial port object based on the object type and the serial port.
Serial Port Descriptive Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the serial port object.</td>
</tr>
<tr>
<td>Port</td>
<td>Indicate the platform-specific serial port name.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for s with the get function.

```matlab
get(s,{'Name','Port','Type'})
an =
    'Serial-COM1'     'COM1'     'serial'
```

**Note** The serial port is not locked by MATLAB, so other applications or other instances of MATLAB can access the same serial port. This might result in a conflict, with unpredictable results.

The Serial Port Object Display

The serial port object provides you with a convenient display that summarizes important configuration and state information. You can invoke the display summary these 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 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.

The display summary for the serial port object s is given below.

```matlab
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
Configuring 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 6-9. These properties are as follows:

**Serial Port Communication Properties**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaudRate</td>
<td>Specify the rate at which bits are transmitted.</td>
</tr>
<tr>
<td>DataBits</td>
<td>Specify the number of data bits to transmit.</td>
</tr>
<tr>
<td>Parity</td>
<td>Specify the type of parity checking.</td>
</tr>
<tr>
<td>StopBits</td>
<td>Specify the number of bits used to indicate the end of a byte.</td>
</tr>
<tr>
<td>Terminator</td>
<td>Specify the terminator character.</td>
</tr>
</tbody>
</table>

**Note** If the serial port object and the instrument communication settings are not identical, you cannot 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 “The Serial Port Object” on page 6-16 with the get function.

```matlab
get(s,{'BaudRate','DataBits','Parity','StopBits',' Terminator'})
ans =
   [9600]    [8]    'none'    [1]    'LF'
```
Writing and Reading Data

Asynchronous Write and Read Operations

Asynchronous write and read operations do not block access to the MATLAB command line. Additionally, while an asynchronous operation is in progress you can

- Execute a read (write) operation while an asynchronous write (read) operation is in progress. This is because serial ports have separate pins for reading and writing.
- Make use of all supported callback properties. Refer to “Events and Callbacks” on page 6-25 for more information about the callback properties supported by serial port objects.

The process of writing data asynchronously is given in “Synchronous Versus Asynchronous Write Operations” on page 3-17.

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 2-7.

Asynchronous Read Operations

For serial port objects, you specify whether read operations are synchronous or asynchronous with the ReadAsyncMode property. You can configure ReadAsyncMode to continuous or manual.

If ReadAsyncMode is continuous (the default value), the serial port object continuously queries the instrument to determine if data is available to be read. If data is available, it is asynchronously stored in the input buffer. To transfer the data from the input buffer to MATLAB, you use one of the
synchronous (blocking) read functions such as fgets, fgets, fscanf, or fread. If data is available in the input buffer, these functions will return quickly.

```matlab
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'continuous';
fprintf(s,'*IDN?')
s.BytesAvailable
ans =
  56
out = fscanf(s);
```

If ReadAsyncMode is manual, the serial port object does not continuously query the instrument to determine if data is available to be read. To read data asynchronously, you use the readasync function. You then use one of the synchronous read functions to transfer data from the input buffer to MATLAB.

```matlab
s.ReadAsyncMode = 'manual';
fprintf(s,'*IDN?')
s.BytesAvailable
ans =
  0
readasync(s)
s.BytesAvailable
ans =
  56
out = fscanf(s);
```

**Rules for Completing Write and Read Operations**

The rules for completing synchronous and asynchronous read and write operations are described below.

**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.
Controlling Instruments Using the Serial Port

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 serial port 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.
- The time specified by the Timeout property passes.
- The input buffer is filled.
- The specified number of values is read (fscanf and readasync only).

A read operation with fread completes when one of these conditions is satisfied:

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

In addition to these rules, you can stop an asynchronous read operation at any time with the stopasync function.

Example: 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 given below are specific to this instrument. A sine wave is input into channel 2 of the
oscilloscope, and your job is to measure the peak-to-peak voltage of the input signal.

1 **Create a serial port object** — Create the serial port object \( s \) associated with serial port COM1.

\[
s = \text{serial('COM1')};
\]

2 **Connect to the instrument** — Connect \( s \) to the oscilloscope. Because the default value for the ReadAsyncMode property is continuous, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

\[
fopen(s)
\]

3 **Write and read data** — Write the *IDN? command to the instrument using fprintf, and then read back the result of the command using fscanf.

\[
\text{fprintf}(s,'*IDN?')
\]
\[
s.\text{BytesAvailable}
\]
\[
an =
\]
\[
56
\]
\[
\text{idn} = \text{fscanf}(s)
\]
\[
idn =
\]
\[
\text{TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04}
\]

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

\[
\text{fprintf}(s,'MEASUREMENT:IMMED:SOURCE?')
\]
\[
source = \text{fscanf}(s)
\]
\[
source =
\]
\[
\text{CH1}
\]
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.

```matlab
fprintf(s,'MEASUREMENT:IMMED:SOURCE CH2')
fprintf(s,'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(s)
source = CH2
```

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

```matlab
fprintf(s,'MEASUREMENT:MEAS1:TYPE PK2PK')
fprintf(s,'MEASUREMENT:MEAS1:VALUE?')
```

Transfer data from the input buffer to MATLAB using `fscanf`.

```matlab
ptop = fscanf(s)
ptop = 2.0199999809E0
```

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

```matlab
fclose(s)
delete(s)
clear s
```
Events and Callbacks

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Event Types and Callback Properties” on page 6-25</td>
</tr>
<tr>
<td>“Storing Event Information” on page 6-26</td>
</tr>
<tr>
<td>“Example: Using Events and Callbacks” on page 6-28</td>
</tr>
</tbody>
</table>

Event Types and Callback Properties
The event types and associated callback properties supported by serial port objects are listed below.

Serial Port Event Types and Callback Properties

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Associated Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Break interrupt</td>
<td>BreakInterruptFcn</td>
</tr>
<tr>
<td>Bytes available</td>
<td>BytesAvailableFcn</td>
</tr>
<tr>
<td></td>
<td>BytesAvailableFcnCount</td>
</tr>
<tr>
<td></td>
<td>BytesAvailableFcnMode</td>
</tr>
<tr>
<td>Error</td>
<td>ErrorFcn</td>
</tr>
<tr>
<td>Output empty</td>
<td>OutputEmptyFcn</td>
</tr>
<tr>
<td>Pin status</td>
<td>PinStatusFcn</td>
</tr>
<tr>
<td>Timer</td>
<td>TimerFcn</td>
</tr>
<tr>
<td></td>
<td>TimerPeriod</td>
</tr>
</tbody>
</table>

The break-interrupt and pin-status events are described below. For a description of the other event types, refer to “Event Types and Callback Properties” on page 4-29.

Break-Interrupt Event
A break-interrupt event is generated immediately after a break interrupt is generated by the serial port. The serial port generates a break interrupt when
the received data has been in an inactive state longer than the transmission time for one character.

This event executes the callback function specified for the BreakInterruptFcn property. It can be generated for both synchronous and asynchronous read and write operations.

**Pin-Status Event**

A pin-status event is generated immediately after the state (pin value) changes for the CD, CTS, DSR, or RI pins. Refer to “Serial Port Signals and Pin Assignments” on page 6-5 for a description of these pins.

This event executes the callback function specified for the PinStatusFcn property. It can be generated for both synchronous and asynchronous read and write operations.

**Storing Event Information**

You can store 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. As described in “Creating and Executing Callback Functions” on page 4-32, 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 12-6 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.

**Serial Port Event Information**

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Field</th>
<th>Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Break interrupt</td>
<td>Type</td>
<td>BreakInterrupt</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>Bytes available</td>
<td>Type</td>
<td>BytesAvailable</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
</tbody>
</table>
## Serial Port Event Information (Continued)

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Field</th>
<th>Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>Type</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td></td>
<td>Data.Message</td>
<td>An error string</td>
</tr>
<tr>
<td>Output empty</td>
<td>Type</td>
<td>OutputEmpty</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-empty hour:minute:second</td>
</tr>
<tr>
<td>Pin status</td>
<td>Type</td>
<td>PinStatus</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td></td>
<td>Data.PinCarrierDetect, ClearToSend, DataSetReady, or RingIndicator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data.PinValue</td>
<td>on or off</td>
</tr>
<tr>
<td>Timer</td>
<td>Type</td>
<td>Timer</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
</tbody>
</table>

The Data field values are as follows:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbsTime</td>
<td>AbsTime is defined for all events, and indicates the absolute time the event occurred. The absolute time is returned using the MATLAB clock format.</td>
</tr>
<tr>
<td>Pin</td>
<td>Pin is used by the pin status event to indicate if the CD, CTS, DSR, or RI pins changed state. Refer to “Serial Port Signals and Pin Assignments” on page 6-5 for a description of these pins.</td>
</tr>
</tbody>
</table>
Controlling Instruments Using the Serial Port

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PinValue</td>
<td>PinValue is used by the pin status event to indicate the state of the CD, CTS, DSR, or RI pins. Possible values are on or off.</td>
</tr>
<tr>
<td>Message</td>
<td>Message is used by the error event to store the descriptive message that is generated when an error occurs.</td>
</tr>
</tbody>
</table>

**Example: Using Events and Callbacks**

This example uses the M-file callback function `instrcallback` to display event-related information to the command line when a bytes-available event or an output-empty event occurs:

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

   ```
   s = serial('COM1');
   ```

2 **Connect to the instrument** — Connect `s` to the Tektronix TDS 210 oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

   ```
   fopen(s)
   ```

3 **Configure properties** — Configure `s` to execute the callback function `instrcallback` when a bytes-available event or an output-empty event occurs.

   ```
   s.BytesAvailableFcnMode = 'terminator';
   s.BytesAvailableFcn = @instrcallback;
   s.OutputEmptyFcn = @instrcallback;
   ```

4 **Write and read data** — Write the `RS232?` command asynchronously to the oscilloscope. This command queries the RS-232 settings and returns
the baud rate, the software flow control setting, the hardware flow control setting, the parity type, and the terminator.

\[
\text{fprintf(s,'RS232?','async')}
\]

\text{instrcallback} is called after the \text{RS232?} command is sent, and when the terminator is read. The resulting displays are shown below.

\text{OutputEmpty event occurred at 17:37:21 for the object: Serial-COM1.}

\text{BytesAvailable event occurred at 17:37:21 for the object: Serial-COM1.}

Read the data from the input buffer.

\[
\text{out = fscanf(s)}
\]

\[
\text{out = 9600;0;0;NONE;LF}
\]

5 \textbf{Disconnect and clean up} — When you no longer need \textit{s}, you should disconnect it from the instrument, and remove it from memory and from the MATLAB workspace.

\[
\text{fclose(s)}
\]

\[
\text{delete(s)}
\]

\[
\text{clear s}
\]

For a general overview of events and callbacks, including how to create and execute callback functions, refer to “Events and Callbacks” on page 4-28.
Using Control Pins

In this section...

| “Control Pins” on page 6-30 |
| “Signaling the Presence of Connected Devices” on page 6-30 |
| “Controlling the Flow of Data: Handshaking” on page 6-33 |

Control Pins

As described in “Serial Port Signals and Pin Assignments” on page 6-5, 9-pin serial ports include six control pins. The properties associated with the serial port control pins are as follows:

Serial Port Control Pin Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataTerminalReady</td>
<td>Specify the state of the DTR pin.</td>
</tr>
<tr>
<td>FlowControl</td>
<td>Specify the data flow control method to use.</td>
</tr>
<tr>
<td>PinStatus</td>
<td>Indicate the state of the CD, CTS, DSR, and RI pins.</td>
</tr>
<tr>
<td>RequestToSend</td>
<td>Specify the state of the RTS pin.</td>
</tr>
</tbody>
</table>

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 PinStatus property. You can specify or monitor the state of the DTR pin with the DataTerminalReady property.

The following example illustrates how these pins are used when two modems are connected to each other.
Example: Connecting Two Modems

This example connects two modems to each other via 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 Create the instrument objects — 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.

   s1 = serial('COM1');
   s2 = serial('COM2');

2 Connect to the instruments — s1 and s2 are connected to the modems. Because the default value for the ReadAsyncMode property is continuous, data is asynchronously returned to the input buffers as soon as it is available from the modems.

   fopen(s1)
   fopen(s2)

Because the default value of the DataTerminalReady property is on, the computer (data terminal) is now ready to exchange data with the modems. 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 PinStatus property.

   s1.Pinstatus
   ans =
       CarrierDetect: 'off'
       ClearToSend: 'on'
       DataSetReady: 'on'
       RingIndicator: 'off'

The value of the DataSetReady field is on because both modems were powered on before they were connected to the objects.

3 Configure properties — Both modems are configured for a baud rate of 2400 bits per second and a carriage return (CR) terminator.

   s1.BaudRate = 2400;
s1.Terminator = 'CR';
s2.BaudRate = 2400;
s2.Terminator = 'CR';

4 Write and read data — Write the `atd` command to the first modem. This command puts the modem “off the hook,” which is equivalent to manually lifting a phone receiver.

    fprintf(s1,'atd')

Write the `ata` command to the second modem. This command puts the modem in “answer mode,” which forces it to connect to the first modem.

    fprintf(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 `PinStatus` property.

    s1.PinStatus
    ans =
        CarrierDetect: 'on'
        ClearToSend: 'on'
        DataSetReady: 'on'
        RingIndicator: 'off'

You can also verify the modem-modem connection by reading the descriptive message returned by the second modem.

    s2.BytesAvailable
    ans =
        25
    out = fread(s2,25);
    char(out)'
    ans =
        ata
        CONNECT 2400/NONE

Now break the connection between the two modems by configuring the `DataTerminalReady` property to off. You can verify that the modems are disconnected by examining the Carrier Detect pin value.
Using Control Pins

```matlab
s1.DataTerminalReady = 'off';
s1.PinStatus
ans =
    CarrierDetect: 'off'
    ClearToSend: 'on'
    DataSetReady: 'on'
    RingIndicator: 'off'
```

5 **Disconnect and clean up** — Disconnect the objects from the modems, and remove the objects from memory and from the MATLAB workspace.

```matlab
fclose([s1 s2])
delete([s1 s2])
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 6-34
- “Software Handshaking” on page 6-34

**Note** Although you might be able to configure your device for both hardware handshaking and software handshaking at the same time, Instrument Control Toolbox 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 “The RTS and CTS Pins” on page 6-8.

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 `PinStatus` property. You can configure or return the RTS pin value with the `RequestToSend` property.

**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. For Instrument Control Toolbox, 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 `RequestToSend` property. In this case, you should configure `FlowControl` to none. If `FlowControl` is hardware, then the `RequestToSend` 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. These characters, known as Xon and Xoff (or XON and XOFF), are described below.
Software Handshaking Characters

<table>
<thead>
<tr>
<th>Character</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xon</td>
<td>17</td>
<td>Resume data transmission.</td>
</tr>
<tr>
<td>Xoff</td>
<td>19</td>
<td>Pause data transmission.</td>
</tr>
</tbody>
</table>

When using 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.

**Example: 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.

```matlab
fprintf(s,'RS232:SOFTF ON')
s.FlowControl = 'software';
```

To pause data transfer, you write the numerical value 19 (Xoff) to the instrument.

```matlab
fwrite(s,19)
```

To resume data transfer, you write the numerical value 17 (Xon) to the instrument.

```matlab
fwrite(s,17)
```
Controlling Instruments Using TCP/IP and UDP

This chapter describes specific features related to controlling instruments that use the TCP/IP or UDP protocols.

TCP/IP and UDP Overview (p. 7-2) A comparison between the TCP/IP and UDP protocols.

Creating a TCP/IP Object (p. 7-4) The TCP/IP object establishes a connection between MATLAB and the remote host.

Creating an UDP Object (p. 7-10) The UDP object establishes a connection between MATLAB and the remote host.

Writing and Reading Data (p. 7-14) Interface-specific issues related to writing and reading data with a TCP/IP or UDP object.

Events and Callbacks (p. 7-21) Enhance your instrument control application using events and callbacks.
TCP/IP and UDP Overview

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). 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.
Creating a TCP/IP Object

TCP/IP Object
You create a TCP/IP object with the tcpip function. tcpip requires the name of the remote host as an input argument. In most cases, you need to specify the remote port value. If you do not specify the remote port, then 80 is used. As described in “Configuring Properties During Object Creation” on page 3-3, you can also configure property values during object creation.

Each TCP/IP object is associated with one instrument. For example, to create a TCP/IP object for a Sony/Tektronix AWG520 Arbitrary Waveform Generator,

```matlab
t = tcpip('sonytekawg.mathworks.com',4000);
```

Note that the port number is fixed and is found in the instrument’s documentation.

The TCP/IP object t now exists in the MATLAB workspace. You can display the class of t with the who command.

```matlab
whos t
Name      Size   Bytes  Class
--------- ------ ------ --------
t        1x1    640    tcpip object

Grand total is 16 elements using 640 bytes
```

Once the TCP/IP object is created, the following properties are automatically assigned values. These general-purpose properties provide information about the TCP/IP object based on the object type, the remote host, and the remote port.
TCP/IP Descriptive Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the TCP/IP object.</td>
</tr>
<tr>
<td>RemoteHost</td>
<td>Specify the remote host.</td>
</tr>
<tr>
<td>RemotePort</td>
<td>Specify the remote host port for the connection.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for object \( t \) with the `get` function.

```matlab
get(t,{'Name','RemoteHost','RemotePort','Type'})
ans =
    [1x31 char]  [1x24 char]    4000    'tcpip'
```

TCP/IP Object Display

The TCP/IP object provides you with a convenient display that summarizes important configuration and state information. You can invoke the display summary these three ways:

- Type the TCP/IP object variable name at the command line.
- Exclude the semicolon when creating a TCP/IP object.
- Exclude the semicolon when configuring properties 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.

The display summary for the TCP/IP object \( t \) is given below.

```
TCP/IP Object : TCP/IP-sonytekawg.mathworks.com

Communication Settings
    RemotePort: 4000
    RemoteHost: sonytekawg.mathworks.com
    Terminator: 'LF'

Communication State
```
Example: Communicating with a Remote Host

In this example, you read a page from the MathWorks Web site using a TCP/IP object:

1 Create and configure an instrument object — First you create a TCP/IP object in MATLAB. Port 80 is the standard port for Web servers.

   t = tcpip('www.mathworks.com', 80);

   By default, the TCP/IP object has an InputBufferSize of 512, which means it can only read 512 bytes at a time. The MathWorks Web page data is much greater than 512 bytes, so you need to set a larger value for this property.

   set(t, 'InputBufferSize', 30000);

2 Connect the object — Next, you open the connection to the server. If the server is not present or is not accepting connections you would get an error here.

   fopen(t);

3 Write and read data — You can now communicate with the server using the functions fprintf, fscanf, fwrite, and fread.

   To ask a Web server to send a Web page, you use the GET command. You can ask for the main Web page from www.mathworks.com using 'GET /'.

   fprintf(t, 'GET /');
Creating a TCP/IP Object

The server receives the command and sends back the Web page. You can see if any data was sent back by looking at the BytesAvailable property of the object.

```matlab
get(t, 'BytesAvailable')
```

Now you can start to read the Web page data. By default, `fscanf` reads one line at a time. You can read lines of data until the BytesAvailable value is 0. Note that you will not see a rendered web page; the HTML file data will scroll by on the screen.

```matlab
while (get(t, 'BytesAvailable') > 0)
    A = fscanf(t),
end
```

4 **Disconnect and clean up** — If you want to do more communication, you can continue to read and write data here. If you are done with the object, close it and delete it.

```matlab
fclose(t);
delete(t);
clear t
```

**Example: Server Drops the Connection**

This example shows what happens when a TCP/IP object loses its connection with a remote server. The server is a Sony/Tektronix AWG520 Arbitrary Waveform Generator (AWG). Its address is `sonytekawg.mathworks.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.

The AWG can drop the connection because it is taken off line, it is powered down, and so on:

1 **Create an instrument object** — Create a TCP/IP object for the AWG.

   ```matlab
t = tcpip('sonytekawg.mathworks.com', 4000);
```

2 **Connect to the instrument** — Connect to the remote instrument.
3 Write and read data — Write a command to the instrument and read back the result.

```matlab
fprintf(t,'*IDN?')
fscanf(t)
ans =
SONY/TEK,AWG520,0,SCPI:95.0 OS:2.0 USR:2.0
```

Assume that the server drops the connection. If you attempt to read from the instrument, a timeout occurs and a warning is displayed.

```matlab
fprintf(t,'*IDN?')
fscanf(t)
```

Warning: A timeout occurred before the Terminator was reached.
(Type "warning off instrument:fscanf:unsuccessfulRead" to suppress this warning.)
```
ans =
'
```

At this point, the object and the instrument are still connected.

```matlab
get(t,'Status')
an =
open
```

If you attempt to write to the instrument again, an error message is returned and the connection is automatically closed.

```matlab
fprintf(t,'*IDN?')
??? Error using ==> fprintf
    Connection closed by RemoteHost. Use FOPEN to connect to RemoteHost.
```

Note that if the TCP/IP object is connected to the local host, the warning message is not displayed. Instead, the error message is displayed following the next read operation after the connection is dropped.
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.

```matlab
fclose(t)
delete(t)
clear t
```
Creating an UDP Object

In this section...

“UDP Object” on page 7-10
“The UDP Object Display” on page 7-11
“Example: Communicating Between Two Hosts” on page 7-12

UDP Object

You create a UDP object with the `udp` function. `udp` does not require the name of the remote host as an input argument. However, if you are using the object to communicate with a specific instrument, you should specify the remote host and the port number. As described in “Configuring Properties During Object Creation” on page 3-3, you can also configure property values during object creation, such as the `LocalPort` property if you will use the object to read data from the instrument.

For example, to create a UDP object associated with the remote host 127.0.0.1, remote port 4012, and local port 3533,

```
u = udp('127.0.0.1', 4012, 'LocalPort', 3533);```

The UDP object `u` now exists in the MATLAB workspace. You can display the class of `u` with the `whos` command.

```
whos u
Name      Size     Bytes    Class
u         1x1       632      udp object
```

Grand total is 12 elements using 632 bytes

When the UDP object is created, the following properties are assigned values based on the values provided to the `udp` function. These general purpose properties provide information about the UDP object.
Creating an UDP Object

### UDP Descriptive Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a descriptive name for the UDP object.</td>
</tr>
<tr>
<td>RemoteHost</td>
<td>Specify the remote host.</td>
</tr>
<tr>
<td>RemotePort</td>
<td>Specify the remote host port for the connection.</td>
</tr>
<tr>
<td>Type</td>
<td>Indicate the object type.</td>
</tr>
<tr>
<td>LocalPort</td>
<td>Specify the local host port, if you are expecting to receive data from the instrument.</td>
</tr>
</tbody>
</table>

You can display the values of these properties for u with the `get` function.

```plaintext
get(u,{'Name','RemoteHost','RemotePort','Type','LocalPort'})
ans =
    'UDP-127.0.0.1'    '127.0.0.1'    [4012]    'udp'    [3533]
```

### The UDP Object Display

The UDP object provides you with a convenient display that summarizes important configuration and state information. You can invoke the display summary these three ways:

- Type the UDP object variable name at the command line.
- Exclude the semicolon when creating a UDP object.
- Exclude the semicolon when configuring properties 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.

The display summary for the UDP object u is given below.

**UDP Object : UDP-127.0.0.1**

**Communication Settings**

- RemotePort: 4012
- RemoteHost: 127.0.0.1
Example: Communicating Between Two Hosts

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. Note that each host regards the other host’s port as the RemotePort:

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 remote host, the remote port on that other host, and the local port to use on the machine where this object is created:

```matlab
u1 = udp('doetom.dhpc', 'RemotePort', 8866, 'LocalPort', 8844)
```

On host doetom.dhpc, create u2:

```matlab
u2 = udp('doejohn.dhpc', 'RemotePort', 8844, 'LocalPort', 8866)
```

2 Connect the objects — Open both UDP objects, so that each can communicate with the other host.

On host doejohn.dhpc, open u1:

```matlab
fopen(u1)
```

On host doetom.dhpc, open u2:
3 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:

```matlab
defprintf(u1, 'Ready for data transfer.')
```

On host doetom.dhpc, read data coming in from the remote host via u2:

```matlab
defscanf(u2)
    ans =
    Ready for data transfer.
```

4 Disconnect and clean up — When you no longer need u1 on host doejohn.dhpc, you should disconnect it and remove it from memory and from the MATLAB workspace.

```matlab
defclose(u1)
defete(u1)
dele(u1)
```

When you no longer need u2, perform a similar cleanup on the host doetom.dhpc.

```matlab
defclose(u2)
defete(u2)
dele(u2)
```
Writing and Reading Data

In this section...

| “Rules for Completing Write and Read Operations” on page 7-14 |
| “Example: Writing and Reading Data with a TCP/IP Object” on page 7-15 |
| “Example: Writing and Reading Data with a UDP Object” on page 7-18 |

Rules for Completing Write and Read Operations
The rules for completing synchronous and asynchronous read and write operations are described below.

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 2-7.

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 (for `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).

In addition to these rules, you can stop an asynchronous read operation at any time with the `stopasync` function.

Example: Writing and Reading Data with a TCP/IP Object

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 MATLAB, 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.mathworks.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.

   \[
   t = \text{tcpip('sonytekawg.mathworks.com',4000)};
   \]

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.

   \[
   \text{set}(t,'OutputBufferSize',3000)
   \]

   You can now connect \( t \) to the instrument.

   \[
   \text{fopen}(t)
   \]

3 **Write and read data** — Since the instrument’s byte order is little-endian, configure the `ByteOrder` property to `littleEndian`.

   \[
   \text{set}(t,'ByteOrder','littleEndian')
   \]

   Create the sine wave data.

   \[
   x = (0:499).*8*\text{pi}/500;
   \]

   \[
   \text{data} = \sin(x);
   \]

   \[
   \text{marker} = \text{zeros(length(data),1)};
   \]

   \[
   \text{marker}(1) = 3;
   \]

   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.

   \[
   \text{fprintf}(t,'\%s',\{'\text{MMEMORY:DATA "sin.wfm",#42544MAGIC 1000' 13 10}\})
   \]

   \[
   \text{fprintf}(t,'\%s',\{'#42500'\})
   \]
Write the sine wave to the instrument.

```matlab
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.

```matlab
fprintf(t, '"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.

```matlab
fclose(t)
set(t, 'InputBufferSize', 3000)
```

Reopen the connection to the instrument.

```matlab
fopen(t)
```

Read the file `sin.wfm` from the function generator.

```matlab
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.

```matlab
fprintf(t, '"MMEMORY:DATA? "sin.wfm" ')
```

Read the file header as ASCII characters.

```matlab
header1 = fscanf(t)
header1 =
#42544MAGIC 1000"
Read the next six bytes, which specify the length of data.

\[
\text{header2} = \text{fscanf(t,}'\%s',6) \\
\text{header2} = \\
\#42500
\]

Read the waveform using \texttt{float32} precision and read the markers using \texttt{uint8} precision. Note that one \texttt{float32} value consists of four bytes. Therefore, the following commands read 2500 bytes.

\[
\begin{align*}
\text{data} & = \text{zeros}(500,1); \\
\text{marker} & = \text{zeros}(500,1); \\
\text{for} & \ i = 1:500, \\
\text{data}(i) & = \text{fread(t,1,'float32');} \\
\text{marker}(i) & = \text{fread(t,1,'uint8');}
\end{align*}
\]

Read the remaining data, which consists of clock information and termination characters.

\[
\begin{align*}
\text{clock} & = \text{fscanf(t);} \\
\text{cleanup} & = \text{fread(t,2);}
\end{align*}
\]

4 Disconnect and clean up — When you no longer need \texttt{t}, you should disconnect it from the host, and remove it from memory and from the MATLAB workspace.

\[
\begin{align*}
\text{fclose(t)} \\
\text{delete(t)} \\
\text{clear t}
\end{align*}
\]

Example: Writing and Reading Data with a UDP Object

This example illustrates how to use 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 **Connect to the instrument** — Connect \( u \) to the echo server.

   \[ fopen(u) \]

3 **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.

```matlab
set(u,'DatagramTerminateMode','off')
data = fscanf(u)
data =
Request Time Request Time
```

4 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.

```matlab
fclose(u)
delete(u)
clear u
```
Events and Callbacks

In this section...

“Event Types and Callback Properties” on page 7-21
“Storing Event Information” on page 7-22
“Example: Using Events and Callbacks” on page 7-24

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

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Associated Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes available</td>
<td>BytesAvailableFcn</td>
</tr>
<tr>
<td></td>
<td>BytesAvailableFcnCount</td>
</tr>
<tr>
<td></td>
<td>BytesAvailableFcnMode</td>
</tr>
<tr>
<td>Datagram received</td>
<td>DatagramReceivedFcn (UDP objects only)</td>
</tr>
<tr>
<td>Error</td>
<td>ErrorFcn</td>
</tr>
<tr>
<td>Output empty</td>
<td>OutputEmptyFcn</td>
</tr>
<tr>
<td>Timer</td>
<td>TimerFcn</td>
</tr>
<tr>
<td></td>
<td>TimerPeriod</td>
</tr>
</tbody>
</table>

The datagram-received event is described below. For a description of the other event types, refer to “Event Types and Callback Properties” on page 4-29.

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.

**Storing Event Information**
You can store 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. As described in “Creating and Executing Callback Functions” on page 4-32, 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 12-6 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**

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Field</th>
<th>Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes available</td>
<td>Type</td>
<td>BytesAvailable</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hour:minute:second</td>
</tr>
<tr>
<td>Datagram received</td>
<td>Type</td>
<td>DatagramReceived</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hour:minute:second</td>
</tr>
<tr>
<td></td>
<td>Data.DatagramAddress</td>
<td>IP address string</td>
</tr>
<tr>
<td></td>
<td>Data.DatagramLength</td>
<td>Number of bytes received as double</td>
</tr>
<tr>
<td></td>
<td>Data.DatagramPort</td>
<td>Port number of sender as double</td>
</tr>
<tr>
<td>Error</td>
<td>Type</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hour:minute:second</td>
</tr>
<tr>
<td></td>
<td>Data.Message</td>
<td>An error string</td>
</tr>
</tbody>
</table>
TCP/IP and UDP Event Information (Continued)

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Field</th>
<th>Field Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output empty</td>
<td>Type</td>
<td>OutputEmpty</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
<tr>
<td>Timer</td>
<td>Type</td>
<td>Timer</td>
</tr>
<tr>
<td></td>
<td>Data.AbsTime</td>
<td>day-month-year hour:minute:second</td>
</tr>
</tbody>
</table>

The Data field values are described below.

**The AbsTime Field**
AbsTime is defined for all events, and indicates the absolute time the event occurred. The absolute time is returned using the MATLAB `clock` format.

```
day-month-year hour:minute:second
```

**The DatagramAddress Field**
DatagramAddress is the IP address of the datagram sender.

**The DatagramLength Field**
DatagramLength is the length of the datagram in bytes.

**The DatagramPort Field**
DatagramPort is the sender's port number from which the datagram originated.

**The Message Field**
Message is used by the error event to store the descriptive message that is generated when an error occurs.
Example: Using Events and Callbacks

This example extends “Example: Communicating Between Two Hosts” on page 7-12 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.

```matlab
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.
```
This chapter describes specific features and actions related to using device objects. The sections are as follows.

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Objects (p. 8-2)</td>
<td>Basic features of device objects</td>
</tr>
<tr>
<td>Creating and Connecting Device Objects (p. 8-5)</td>
<td>General procedure for the creation and use of device objects</td>
</tr>
<tr>
<td>Communicating with Instruments (p. 8-9)</td>
<td>Object-specific issues related to configuring and exchanging data through device objects</td>
</tr>
<tr>
<td>Device Groups (p. 8-14)</td>
<td>Obtaining and using group device properties</td>
</tr>
</tbody>
</table>
Device Objects

In this section...

“Device Objects: Overview” on page 8-2
“What Are Device Objects?” on page 8-2
“Device Objects for MATLAB Instrument Drivers” on page 8-3

Device Objects: Overview

All instruments attached to your computer must communicate through an interface. Popular interface protocols include GPIB, VISA, and RS-232 (serial). While Instrument Control 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 chapter 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 8-5. If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to Chapter 2, “The Instrument Control Session”.

What Are Device Objects?

Device objects are used to represent instruments in MATLAB. 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 prompt, inside M-Functions, M-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 VXIplug&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 VXIplug&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 Chapter 11, “Using Generic Instrument Drivers”.
Instrument Control Toolbox supports the following interface objects:

- gpib
- serial
- tcpip
- udp
- visa

To learn how to create and use interface objects, see Chapter 3, “Using Interface Objects”.

**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 Instrument Control Toolbox. 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 prompt.

Many other drivers are available on the MathWorks Web site at

```
http://www.mathworks.com/matlabcentral/fileexchange
```

including drivers specifically for Instrument Control Toolbox.
Creating and Connecting Device Objects

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 GBIB 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.

   \[
g = \text{gpib('ni',0,1)};
\]

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.

   \[
d = \text{icdevice('tektronix_tds210', g)};
\]

You can use the whos command to display the size and class of d.

   \[
\text{whos d}
\]

   \[
\begin{array}{llll}
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} \\
\hline
\text{d} & 1x1 & 652 & \text{icdevice object}
\end{array}
\]

   \[
\text{Grand total is 22 elements using 652 bytes}
\]

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.

You can display the current values of all properties for `d` with the `get` command.

```matlab
get(d)
```

**The 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 VXIplug&play, IVI-C, or IVI-COM driver, saving the new driver in a file on disk. The syntax is

```
makemid('driver','filename')
```

where `driver` is the original VXIplug&play, IVI-C, or IVI-COM 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 VXIplug&play, IVI-C, or directly from an IVI-COM driver,

```
obj = icdevice('ConvertedDriver.mdd','GPIB0::2::INSTR')
```

If the driver is created from an IVI-COM logical name,

```
obj = icdevice('ConvertedDriver.mdd')
```

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

```javascript
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.

```javascript
get(d, 'status')
```

```javascript
ans =

open```
Communicating with Instruments

In this section...

“Configuring Instrument Settings” on page 8-9
“Calling Device Object Methods” on page 8-10
“Control Commands” on page 8-12

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.

Example: 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.

```matlab
  g = gpib('ni',0,1);
```
Using Device Objects

```matlab
    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 TriggerSlope settings — Use the get function to
obtain the current value for the oscilloscope TriggerSlope property.

    get(d, 'TriggerSlope')

    ans =

    rising

   The TriggerSlope is currently set to rising.

4 Change the TriggerSlope 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.

    set(d, 'TriggerSlope', 'falling');

   This changes TriggerSlope 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;
```

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.

**Example: 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.

\[
\text{invoke}(d, \text{'measure'}, 1)
\]

\[
\text{ans} = 999.9609
\]

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.

\[
\text{disconnect}(d);
\]
\[
\text{delete}(d);
\]
\[
\text{clear } d g;
\]

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 8-13
- “devicereset” on page 8-13
- “selftest” on page 8-13
- “geterror” on page 8-13

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,

```matlab
get(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.

```matlab
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.

```matlab
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.

```matlab
msg = geterror(obj);
```
Device Groups

In this section...

“Working with Group Objects” on page 8-14
“Example: Using Device Groups to Access Instrument Data” on page 8-15

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.

```matlab
chans = get(d, 'Channel')
```

<table>
<thead>
<tr>
<th>HwIndex:</th>
<th>HwName:</th>
<th>Type:</th>
<th>Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CH1</td>
<td>scope-channel</td>
<td>Channel1</td>
</tr>
<tr>
<td>2</td>
<td>CH2</td>
<td>scope-channel</td>
<td>Channel2</td>
</tr>
</tbody>
</table>

To display the functions that a device group object supports, use the `methods` function.

```matlab
methods(chans(1))
```

You can also display a list of the group object’s properties and their current settings with the `get` function.

```matlab
get(chans(2))
```

To get help on a driver-specific property or function, use the `instrhelp` function, with the name of the function or property.

```matlab
instrhelp(chans(1), 'Coupling')
```
Example: 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 = get(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 VXIplug&play Drivers

This chapter describes the use of VXIplug&play drivers for instrument control. The sections are as follows.

Overview (p. 9-2) Standard VXIplug&play drivers and the configurations required for them.

VXIplug&play Drivers (p. 9-4) Determining whether your system is configured to use VXIplug&play drivers, creating MATLAB VXIplug&play instrument drivers, and creating device objects.
Overview

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“The Instrument Control Toolbox and VXIplug&amp;play Drivers” on page 9-2</td>
</tr>
<tr>
<td>“VISA Setup” on page 9-2</td>
</tr>
<tr>
<td>“Other Software Requirements” on page 9-3</td>
</tr>
</tbody>
</table>

**The Instrument Control Toolbox and VXIplug&play Drivers**

Instrument Control Toolbox can communicate with hardware using VXIplug&play drivers. Most often, the instrument manufacturers supply these drivers.

For definitions and specifications of VXIplug&play drivers, see the Web site of the VXIplug&play Systems Alliance at http://www.vxipnp.org.

**VISA Setup**

A system must have VISA installed in order for VXIplug&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 prompt, 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 `agilent`, `tek`, 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 VXIplug&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**

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Installing VXI plug&amp;play Drivers” on page 9-4</td>
</tr>
<tr>
<td>“Creating a MATLAB VXIplug&amp;play Instrument Driver” on page 9-5</td>
</tr>
<tr>
<td>“Constructing Device Objects Using a MATLAB VXIplug&amp;play Instrument Driver” on page 9-8</td>
</tr>
</tbody>
</table>

**Installing VXI plug&play Drivers**

The VXIplug&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 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 VXIplug&play drivers will be listed.

Alternatively, you can use the MATLAB command-line function `instrhwinfo` to find out which drivers are installed.
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 VXI plug&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 Web Site**

You might find an appropriate MATLAB driver wrapper for your instrument on the MathWorks Web site, on the Supported Hardware page for Instrument Control Toolbox, at


On this page, click the Instrument Drivers link. You then have a choice to go to the MATLAB Central File Exchange, where you can look for the driver you need, or you can submit a request to the MathWorks for your particular driver with the Instrument Driver Request Form.

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 Web site.

**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 VXIplug&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 VXIplug&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 VXIplug&play driver, thereby creating a MATLAB VXIplug&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 VXIplug&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.
With the MATLAB Instrument Driver Editor, you can:

- Create, delete, modify, and rename properties, functions, or groups.
• Add M-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 Chapter 14, “Using the Instrument Driver Editor”.

**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 VXI plug&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:

```matlab
obj = icdevice('MATLABVXIpnpDriver.mdd','GPIB::2::INSTR')
connect(obj)
```

See the `icdevice` reference page for full details about this function.
Using IVI Drivers

This chapter describes the use of IVI drivers for instrument control. The sections are as follows.

- IVI Drivers Overview (p. 10-2)  VISA or other requirements for your system to use IVI drivers.
- IVI Drivers (p. 10-4)  Examine your IVI driver installation. Create MATLAB IVI instrument drivers. Create device objects for MATLAB IVI instrument drivers.
- IVI Configuration Store (p. 10-11)  Examine and configure your IVI configuration store.
IVI Drivers Overview

In this section...

“Instrument Control Toolbox and IVI Drivers” on page 10-2
“VISA Setup” on page 10-2
“IVI Shared Components” on page 10-3
“IVI Configuration Store Overview” on page 10-3

Instrument Control Toolbox and IVI Drivers

Instrument Control Toolbox can communicate with instruments using Interchangeable Virtual Instrument (IVI) drivers. The toolbox supports both IVI-C and IVI-COM drivers, provided by various instrument manufacturers.

For definitions and specifications of IVI drivers and their components, see the Web site of the IVI Foundation at http://www.ivifoundation.org.

VISA Setup

Most IVI drivers require that VISA be installed on your system. 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 prompt, 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 agilent, tek, and ni.

If you need to install VISA, the software installation disk provided with your instrument might include VISA along with the instrument’s IVI driver, or you might be able to download VISA from the instrument manufacturer’s Web site.
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.

**IVI Shared Components**

Many driver elements are common to a wide variety of instruments, so are not contained in the driver itself. Instead, these elements are delivered as *shared components*. Sharing components keeps the drivers themselves as small and interchangeable as possible. You can use `instrhwinfo` to determine whether shared components are installed on your system; see “Installing IVI Drivers” on page 10-5.

**IVI Configuration Store Overview**

An IVI configuration store is a configured arrangement of instrument drivers and I/O port references. For an overview of IVI configuration stores and their components, see “IVI Configuration Store” on page 10-11.
**IVI Drivers**

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“MATLAB IVI Driver” on page 10-4</td>
</tr>
<tr>
<td>“IVI-C and IVI-COM” on page 10-4</td>
</tr>
<tr>
<td>“Installing IVI Drivers” on page 10-5</td>
</tr>
<tr>
<td>“MATLAB IVI Instrument Drivers” on page 10-6</td>
</tr>
<tr>
<td>“Constructing Device Objects Using a MATLAB IVI Instrument Driver” on page 10-9</td>
</tr>
</tbody>
</table>

**MATLAB IVI Driver**

After your IVI driver is installed, you create a MATLAB IVI driver based on the IVI driver. With the MATLAB IVI driver, you construct a device object, which you use to communicate with your instrument.

**IVI-C and IVI-COM**

Instrument Control Toolbox supports both IVI-C and IVI-COM drivers, with class-compliant and instrument-specific functionality. IVI class-compliant drivers are those that support common functionality across a family of related instruments. Class-compliant drivers might be appropriate if you want to use the basic functionality of an instrument, with the ability to swap instruments without changing the code in your application. An IVI instrument-specific driver or interface provides access to the instrument’s unique functionality, and therefore generally does not accommodate instrument substitution.

In the case of IVI-C drivers, you can use IVI-C class drivers and IVI-C specific drivers. Device objects that are constructed to call IVI-C class drivers offer interchangeability between similar instruments, and will work with all instruments consistent with that class driver. Device objects constructed to directly call IVI-C specific drivers generally offer less interchangeability, but provide access to the unique methods and properties of a specific instrument.

Device objects that are constructed to use IVI-COM drivers might have access to both class-compliant and instrument-specific functionality. Most IVI-COM drivers contain instrument-specific properties and methods. Many also
contain properties and methods that comply with IVI class specifications. This allows you to choose between interchangeability and instrument-specific functionality with the same driver. An application that uses only the class-compliant functionality of a driver works with all drivers that comply with the same IVI class. Applications that employ instrument-specific functionality might work only with that driver on that instrument.

**Installing IVI Drivers**

The IVI driver particular to a piece of equipment is usually provided by the equipment manufacturer, either on an installation disk or as a Web download. Required VISA software and IVI shared components might also come with the driver.

You can use the MATLAB command-line function `instrhwinfo` to find information on installed IVI drivers and shared components.

```matlab
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\'
```

LogicalNames are associated with particular IVI drivers, but they do not necessarily imply that the drivers are currently installed. Drivers can be installed that do not have a LogicalName property set yet, or drivers that had LogicalName set might have been removed.

ConfigurationServerVersion, MasterConfigurationStore, and IVIRootPath all convey information related to installed shared components. The ConfigurationServerVersion indicates whether IVI shared components are installed. If its value is an empty string, then no shared components were found.
Alternatively, you can 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 10-11.)

**MATLAB IVI Instrument Drivers**

To use an IVI driver with a device object, you must have 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. You can download or create the MATLAB instrument driver.
**Downloading a Driver from the MathWorks Web Site**

You might find an appropriate MATLAB driver wrapper for your instrument on the MathWorks Web site, on the Supported Hardware page for Instrument Control Toolbox, at


On this page, click the **Instrument Drivers** link. You then have a choice to go to the **MATLAB Central File Exchange**, where you can look for the driver you need, or you can submit a request to the MathWorks for your particular driver with the **Instrument Driver Request Form**.

To use the downloaded MATLAB IVI driver, you must also have the instrument's IVI driver installed. This driver might be available from the instrument manufacturer's Web site.

**Creating a 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')
```

where `driver` is the original IVI driver name (identified by `instrhwinfo` or the Test & Measurement Tool), and `filename` is the file containing the conversion result. For `driver` you can use a **Module** (for IVI-C), a **ProgramID** (for IVI-COM), or a **LogicalName** (for either IVI-C or IVI-COM). See the `makemid` reference page for full 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 an IVI 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 an IVI driver, thereby creating a MATLAB IVI 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 by typing `midedit`.
2. Select **File > Import**.
3. In the Import Driver dialog box, select the IVI driver that you want to import, and click **Import**.

The MATLAB Instrument Driver Editor loads the driver and displays its components, as shown in the following figure.

![MATLAB Instrument Driver Editor Showing TekScope MATLAB Instrument Driver Display Group Functions](image)

With the MATLAB Instrument Driver Editor, you can do the following:

- Create, delete, modify, and rename properties, functions, or groups.
- Add M-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 Chapter 14, “Using the Instrument Driver Editor”.

**Note** When you create a MATLAB instrument driver based on an IVI 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 IVI Instrument Driver

Once you have the MATLAB IVI instrument driver, you create the device object with the filename of the MATLAB instrument driver as an argument for `icdevice`. The following examples show the creation of MATLAB IVI instrument drivers (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 `LogicalName` to identify an IVI driver, then creates a MATLAB IVI driver. Because `LogicalName` is associated with a driver session and hardware asset, you do not need to pass a `RsrcName` to `icdevice` when constructing the device object.

```matlab
makemid('MainScope','MainScope.mdd')
obj = icdevice('MainScope.mdd')
```

In the next example, `makemid` uses a `ProgramID` to reference an IVI-COM driver, then creates a MATLAB IVI instrument driver. The device object requires a `RsrcName` in addition to the filename of the MATLAB IVI instrument driver.

```matlab
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 filename of the MATLAB IVI instrument driver.

```matlab
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 filename and the logical name of the driver from the IVI configuration store.

```matlab
makemid('ivifgen','FgenML.mdd')
obj = icdevice('FgenML.mdd','FuncGen')
```
**Benefits of an IVI Configuration Store**

By providing a way outside of the application to configure the relationship between drivers and I/O interface references, an IVI configuration store greatly enhances instrument interchangeability.

Suppose your application 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 application to accommodate that change.

An IVI configuration store offers the ability to accommodate different instrument models, drivers, or ports, without having to modify your application. This is especially useful when using a compiled application that you cannot easily modify.

**Components of an IVI Configuration Store**

The components of the IVI configuration store identify the locations of the instruments to communicate with, the software modules used to control the instruments, and the associations of which software modules are used with instruments at which locations.
### Component Description

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Module</td>
<td>A software module is instrument-specific, and contains the commands and functions necessary to communicate with the instrument. Software modules are commonly provided by the instrument vendor, and they are not editable from within MATLAB.</td>
</tr>
<tr>
<td>Hardware Asset</td>
<td>A hardware asset is an identification of a communication port to which the instrument is connected. This component is configured with an IOResourceDescriptor. Usually you have one hardware asset per connection location (protocol type, bus address, etc.).</td>
</tr>
<tr>
<td>Driver Session</td>
<td>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 that are connected to the same location (at different times) can use the same hardware asset, but because they have different software modules, they require different driver sessions.</td>
</tr>
<tr>
<td>Logical Name</td>
<td>A logical name is a configuration store component that provides access to a driver session. A logical name can be thought of as a configurable pointer to a driver session. In a typical setup, the application communicates with an instrument via a logical name. If the application needs to communicate with a different instrument (for example, the same kind of scope at a different location), only the logical name within the IVI configuration store needs to be updated to point to the new driver session; there is no need to rewrite any code in the application, because it uses the same logical name.</td>
</tr>
</tbody>
</table>
Configuring an IVI Configuration Store

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`. 
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.

```matlab
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.

```matlab
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: ''
```

The configuration of this example is shown below, used by the application data_analyzer.m.

You create and configure elements in the IVI configuration store by using the IVI configuration store object methods add, commit, remove, and update. For further details, see the reference pages for these methods.
Example: Basic IVI Configuration Store

The following figure shows an example of an IVI configuration store with several interchangeable components. Application 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. Because the scopes are at different locations (or perhaps it is the same scope connected to two different locations at different times), each configuration requires its own driver session, in this example, driver session A and driver session B.

The code of application 1 is written to access logical name 1, which is configured in the IVI configuration store to access driver session A or driver session B (but not both at the same time). Because the selection of driver session is performed in the IVI configuration store, there is no need to alter the code of the application to change access from one scope to the other.
IVI Configuration Store with Several Interchangeable Elements
This chapter describes the use of generic drivers for controlling instruments from MATLAB with Instrument Control Toolbox. The sections are as follows.

- **Generic Drivers: Overview (p. 11-2)** Introduces generic instrument drivers.
- **Example: Writing a Generic Driver (p. 11-3)** Illustrates how to write a generic instrument driver with the MATLAB Instrument Driver Editor.
- **Example: Using a Generic Driver with the Test & Measurement Tool (p. 11-9)** Illustrates how to use a generic driver with the Test & Measurement Tool GUI.
- **Example: Using a Generic Driver at the Command Line (p. 11-13)** Illustrates how to use a generic driver with MATLAB commands that work at the command line or in M-code.
Generic Drivers: Overview

Generic drivers allow Instrument Control Toolbox 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 MATLAB), 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:

- “Example: Writing a Generic Driver” on page 11-3
- “Example: Using a Generic Driver with the Test & Measurement Tool” on page 11-9
- “Example: Using a Generic Driver at the Command Line” on page 11-13
Example: Writing a Generic Driver

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Creating the Driver and Defining Its Initialization Behavior” on page 11-3</td>
</tr>
<tr>
<td>“Defining Properties” on page 11-5</td>
</tr>
<tr>
<td>“Defining Functions” on page 11-8</td>
</tr>
</tbody>
</table>

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 Instrument Control Toolbox 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 line.

   `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.

   ![MATLAB Instrument Driver Editor](image)
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:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer</td>
<td>Microsoft</td>
</tr>
<tr>
<td>Supported models</td>
<td>IE</td>
</tr>
<tr>
<td>Instrument type</td>
<td>Browser</td>
</tr>
<tr>
<td>Driver version</td>
<td>1.0</td>
</tr>
</tbody>
</table>

5 Select the node **Initialization and Cleanup**.

6 Click the **Create** tab.

This is where you define the M-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:

```matlab
ie = actxserver('internetexplorer.application');
set(obj, 'Interface', ie);
```

8 Click the **Connect** tab.

This is where you define the M-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:

```matlab
ie = get(obj, 'Interface');
set(ie, 'Visible', 1);
set(ie, 'FullScreen', 0);
```
The first line gets \texttt{ie} as a handle to the COM object, based on the assignment in the \texttt{Create} code. The two lines after that set the window visibility and size.

\section*{Defining Properties}

Writing properties for generic drivers in the MATLAB Instrument Driver Editor is a matter of writing straight M-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.

\subsection*{Using the Same Name for a Property}

The position of the IE browser window is determined by the \texttt{Top} and \texttt{Left} properties of its COM object. In the following steps, you make the \texttt{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 \texttt{Properties} node in the driver editor tree.

2. In the \texttt{Add property} field, enter the text \texttt{Top}, and click \texttt{Add}.

3. Expand the \texttt{Properties} node in the tree, and select the new node \texttt{Top}.

4. Click the \texttt{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 \texttt{Data Type} field indicates \texttt{Double}. In the \texttt{Constraint} field, click the pull-down menu and select \texttt{Bounded}.

6. Keep the \texttt{Minimum} value of 0.0, and enter a \texttt{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:

```matlab
ie = get(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:

```javascript
ie = get(obj, 'Interface');
set(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:

```javascript
ie = get(obj, 'Interface');
propertyValue = get(ie, 'Height');
```

9 Add the following code at the bottom of the function in the **Set code** pane:

```javascript
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 property 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 M-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 string 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 create 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 = get(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 & Measurement Tool (tmtool) or at the MATLAB command line.
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 “Example: Writing a Generic Driver” on page 11-3.

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.
Example: Using a Generic Driver at the Command Line

In this section...

“Creating and Connecting the Device Object” on page 11-13
“Accessing Properties” on page 11-14
“Using Functions” on page 11-15

Creating and Connecting the Device Object

Instrument Control Toolbox provides MATLAB commands you can use in the Command Window or in M-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 “Example: Writing a Generic Driver” on page 11-3.

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.

    ```matlab
    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

    ```matlab
    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.

    ```matlab
    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 MATLAB.

**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.

```plaintext
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
```
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.

```matlab
set(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.

```matlab
get(ie_obj, 'Top')
ans =
120
```

Adjust the size of the window by setting the Vsize property.

```matlab
set(ie_obj, 'Vsize', 200);
```

6 Make the window larger by increasing the property value.

```matlab
set(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:

<table>
<thead>
<tr>
<th>Contents</th>
<th>disp</th>
<th>icdevice</th>
<th>instrnotify</th>
<th>methods</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>display</td>
<td>igetfield</td>
<td>instrument</td>
<td>ne</td>
<td>subsasgn</td>
</tr>
<tr>
<td>close</td>
<td>end</td>
<td>inspect</td>
<td>invoke</td>
<td>obj2mfile</td>
<td>subsref</td>
</tr>
<tr>
<td>connect</td>
<td>eq</td>
<td>instrcallback</td>
<td>isa</td>
<td>open</td>
<td>vertcat</td>
</tr>
<tr>
<td>ctranspose</td>
<td>fieldnames</td>
<td>instrfind</td>
<td>isequal</td>
<td>openvar</td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>get</td>
<td>instrfindall</td>
<td>isetfield</td>
<td>propinfo</td>
<td></td>
</tr>
<tr>
<td>devicereset</td>
<td>geterror</td>
<td>instrhelp</td>
<td>isValid</td>
<td>selftest</td>
<td></td>
</tr>
<tr>
<td>disconnect</td>
<td>horzcat</td>
<td>instrhwinfo</td>
<td>length</td>
<td>set</td>
<td></td>
</tr>
</tbody>
</table>

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.
## Saving and Loading the Session

This chapter describes how to save and load information associated with an instrument control session. The sections are as follows.

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saving and Loading Instrument Objects (p. 12-2)</td>
<td>Save instrument objects and their associated property values to disk as an M-file or as a MAT-file.</td>
</tr>
<tr>
<td>Debugging: Recording Information to Disk (p. 12-6)</td>
<td>Save information to disk as a text file. The saved information includes the data transferred to and from the instrument, and event information.</td>
</tr>
</tbody>
</table>
Saving and Loading Instrument Objects

In this section...

“Saving Instrument Objects to an M-File” on page 12-2
“Saving Objects to a MAT-File” on page 12-4

Saving Instrument Objects to an M-File

You can save an instrument object to an M-file using the \texttt{obj2mfile} function. \texttt{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 \texttt{propinfo} function or examine the property reference pages.

- Save property values using the \texttt{set} syntax or the dot notation.

  If the \texttt{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 M-file containing the instrument object code.

For example, suppose you create the GPIB object \texttt{g}, return instrument identification information to the variable \texttt{out}, and store \texttt{out} in the \texttt{UserData} property.

\begin{verbatim}
g = gpib('ni',0,1); g.Tag = 'My GPIB object'; fopen(g) cmd = '*IDN?'; fprintf(g,cmd) out = fscanf(g); g.UserData = out;
\end{verbatim}
The following command saves g and the modified property values to the M-file mygpib.m. Because the UserData property is not empty, its value is automatically written to the MAT-file mygpib.mat.

```matlab
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 an M-file into the MATLAB workspace, type the name of the M-file at the command line. For example, to load g from the M-file mygpib.m,

```matlab
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.

```matlab
g.UserData
ans =
```
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 an M-File” on page 12-2, 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 12-6
“Example: Introduction to Recording Information” on page 12-7
“Creating Multiple Record Files” on page 12-7
“Specifying a File Name” on page 12-8
“Record File Format” on page 12-8
“Example: Recording Information to Disk” on page 12-10

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

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordDetail</td>
<td>Specify the amount of information saved to a record file.</td>
</tr>
<tr>
<td>RecordMode</td>
<td>Specify whether data and event information are saved to one record file or to multiple record files.</td>
</tr>
</tbody>
</table>
Recording Properties (Continued)

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecordName</td>
<td>Specify the name of the record file.</td>
</tr>
<tr>
<td>RecordStatus</td>
<td>Indicate if data and event information are saved to a record file.</td>
</tr>
</tbody>
</table>

Example: 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`.

```plaintext
g = gpib('ni',0,1);
g.RecordName = 'myfile.txt';
fopen(g)
record(g)
fprintf(g,'*IDN?')
out = fscanf(g);
```

End the instrument control session.

```plaintext
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 12-8.

**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

\[
\text{value} = (-1)^{\text{sign}} \times 2^{\text{exp} - 127} \times (1.\text{significand})
\]

Double-precision floating-point values consist of 64 bits, and the value is given by

\[
\text{value} = (-1)^{\text{sign}} \times 2^{\text{exp} - 1023} \times (1.\text{significand})
\]

The floating-point format component and the associated single-precision and double-precision bits are given below.

<table>
<thead>
<tr>
<th>Format Component</th>
<th>Single-Precision Bits</th>
<th>Double-Precision Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>exp</td>
<td>2-9</td>
<td>2-12</td>
</tr>
<tr>
<td>significand</td>
<td>10-32</td>
<td>13-64</td>
</tr>
</tbody>
</table>

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 \texttt{hex2dec}:

\[
dval = \text{hex2dec}'40880000'
dval = 1.082654720000000e+009
\]

Convert the decimal value to a binary value using \texttt{dec2bin}:

\[
bval = \text{dec2bin}(dval,32)
bval = 01000000100010000000000000000000
\]

The interpretation of \texttt{bval} is given by the preceding table. The leftmost bit indicates the value is positive because \((-1)^0 = 1\). The next 8 bits correspond to the exponent, which is given by

\[
\text{exp} = \text{bval}(2:9)
\]
exp =
10000001

The decimal value of exp is $2^7 + 2^0 = 129$. The remaining bits correspond to the significand, which is given by

```
significand = bval(10:32)
significand =
000100000000000000000000
```

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
```

**Example: Recording Information to Disk**

This example extends “Example: Reading Binary Data” on page 4-22 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.

   ```matlab
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.

   ```matlab
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.

\[
\begin{align*}
g.\text{RecordMode} &= 'index'; \\
g.\text{RecordDetail} &= 'verbose'; \\
g.\text{RecordName} &= 'WaveForm1.txt';
\end{align*}
\]

3 Connect to the instrument — Connect g to the oscilloscope.

\[
fopen(g)
\]

4 Write and read data — Initiate recording.

\[
\text{record}(g)
\]

Configure the scope to transfer the screen display as a bitmap.

\[
\begin{align*}
fprintf(g, 'HARDCOPY:PORT GPIB') \\
fprintf(g, 'HARDCOPY:FORMAT BMP') \\
fprintf(g, 'HARDCOPY START')
\end{align*}
\]

Initiate the asynchronous read operation, and begin generating events.

\[
\text{readasync}(g)
\]

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 MATLAB as unsigned 8-bit integers.

\[
\text{out} = \text{fread}(g, g.\text{BytesAvailable}, 'uint8');
\]

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
* 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
13 Recording off.
This chapter describes how to use the Test & Measurement Tool to access your hardware interfaces and instrument drivers.

**Test & Measurement Tool: Overview**  
(p. 13-2)  
Describes the capabilities of the Test & Measurement Tool, its appearance, and how to start it up.

**Using the Test & Measurement Tool**  
(p. 13-4)  
Describes through examples how to use the Test & Measurement Tool to communicate with an instrument.
Test & Measurement Tool: Overview

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Instrument Control Toolbox Support” on page 13-2</td>
</tr>
<tr>
<td>“Navigating the Tree” on page 13-3</td>
</tr>
</tbody>
</table>

Instrument Control Toolbox 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 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 MATLAB script.
- Visualize acquired data.
- Export acquired data to the MATLAB environment.
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.
Using the Test & Measurement Tool

In this section...

“Overview of the Examples” on page 13-4
“Hardware” on page 13-4
“Instrument Objects” on page 13-12
“Instrument Drivers” on page 13-16

Overview of the Examples

These examples illustrate 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 line, 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.

![Scan Result Image]
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 4.
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 15 seconds.
Establishing the Connection

Expand the keithley-Board-0 node and select the instrument at primary address 4: PAD-4 (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
- MATLAB array 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.

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
- M-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 an M-file. (When you run that M-file, it creates a new object with the equivalent settings.)

![Object Exporter](image)

**Saving Your Instrument Control Session.** The Session Log tab displays the M-code equivalent of your instrument control session. You can save this code to an M-file so that you can execute the same commands programmatically.

Select **File > Save Session Log** from the menu bar or click **Save Session** on the Session Log pane. This opens the Export to M-File dialog box, which enables you to specify a file name and directory location for the M-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 line, 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 path.
When the Test & Measurement Tool scans for drivers, it makes them available as nodes under the driver type node. Expand the MATLAB Instrument Drivers node to reveal the installed drivers. Select one of them to see the driver’s details.

![Image of Test & Measurement Tool]

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 XVIplug&play drivers with the Test & Measurement Tool, see “VXIplug&play Drivers” on page 9-4.

**IVI Drivers**

For an example of scanning for installed IVI-C or IVI-COM drivers with the Test & Measurement Tool, see “IVI Drivers” on page 10-4. For using the Test & Measurement Tool to examine or configure an IVI configuration store, see “Configuring an IVI Configuration Store” on page 10-13.
Using the Instrument Driver Editor

This chapter describes how to use the Instrument Driver Editor to create, import, or modify instrument drivers. The sections are as follows.

MATLAB Instrument Driver Editor: Overview (p. 14-2)  An explanation of MATLAB instrument drivers, their features, capabilities, and usage.

Creating MATLAB Instrument Drivers (p. 14-5)  How to create, modify, save, and verify instrument drivers using the MATLAB Instrument Driver Editor.

Properties (p. 14-18)  Defining instrument functionality as properties of the device object to ease communications with your instrument.

Functions (p. 14-34)  Defining instrument functionality as functions of the device object to ease communications with your instrument.

Groups (p. 14-46)  Combining common capabilities of the device object.

Using Existing Drivers (p. 14-65)  Adapting existing drivers for your instrument.
MATLAB Instrument Driver Editor: Overview

In this section...

“What Is a MATLAB Instrument Driver?” on page 14-2
“How Does a MATLAB Instrument Driver Work?” on page 14-3
“Why Use a MATLAB Instrument Driver?” on page 14-3

What Is a MATLAB Instrument Driver?

Instrument Control Toolbox 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 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 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 14-65.

### 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 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

Creating MATLAB Instrument Drivers

In this section...

“Driver Components” on page 14-5
“MATLAB Instrument Driver Editor Features” on page 14-6
“Saving MATLAB Instrument Drivers” on page 14-6
“Driver Summary and Control Commands” on page 14-6
“Initialization and Cleanup” on page 14-10

Driver Components

A MATLAB instrument driver contains information about an instrument and defines the functionality supported by the instrument.

<table>
<thead>
<tr>
<th>Driver Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver Summary and Control Commands</td>
<td>Basic information about the instrument, e.g., manufacturer or model number.</td>
</tr>
<tr>
<td>Initialization and Cleanup</td>
<td>Code that is executed at various stages in the instrument control session, e.g., code that is executed upon connecting to the instrument.</td>
</tr>
<tr>
<td>Properties</td>
<td>A property is generally used to configure or query an instrument’s state information.</td>
</tr>
<tr>
<td>Functions</td>
<td>A function is generally used to control or configure an instrument.</td>
</tr>
<tr>
<td>Groups</td>
<td>A group combines common functionality of the instrument into one component.</td>
</tr>
</tbody>
</table>

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.
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 MATLAB code to wrap around commands sent to instrument.

You can open the MATLAB Instrument Driver Editor with the `midedit` command.

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 Control 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. Control 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 14-7
- “Control Commands” on page 14-7
• “Example: Defining Driver Summary and Control Commands” on page 14-7
• “Verifying Driver Summary and Control Commands” on page 14-9

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.

Control Commands
You can define basic control commands supported by the instrument. The control commands can be accessed through device object properties and functions.

<table>
<thead>
<tr>
<th>Control Command</th>
<th>Accessed with Device Object’s</th>
<th>Example Instrument Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify</td>
<td>InstrumentModel property</td>
<td>*IDN?</td>
<td>Returns the identification string of the instrument</td>
</tr>
<tr>
<td>Reset</td>
<td>devicereset function</td>
<td>*RST</td>
<td>Returns the instrument to a known state</td>
</tr>
<tr>
<td>Self test</td>
<td>selftest function</td>
<td>*TST?</td>
<td>Tests the instrument’s interface</td>
</tr>
<tr>
<td>Error</td>
<td>geterror function</td>
<td>ErrLog:Next?</td>
<td>Retrieves the next instrument error message</td>
</tr>
</tbody>
</table>

The MATLAB Instrument Driver Editor assigns default values for the control commands. The control commands should be modified appropriately to match the instrument’s command set.

Example: Defining Driver Summary and Control Commands
This example defines the basic driver information and control commands for a Tektronix TDS 210 oscilloscope using the MATLAB Instrument Driver Editor.

1 Select the Summary node in the tree.
Using the Instrument Driver Editor

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 **Control 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 14-6.

Verifying Driver Summary and Control Commands

This procedure verifies the summary information defined in the Driver Summary and Control 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 Keithley GPIB board at board index 0. From the MATLAB command line,

1. Create the device object, `obj`, using the `icdevice` function.

   ```matlab
g = gpib('keithley', 0, 2);
obj = icdevice('tektronix_tds210_ex.mdd', g);
```

2. View the defined driver information.

   ```matlab
   obj
   ```

   Instrument Device Object Using Driver : `tektronix_tds210_ex.mdd`

   Instrument Information
   Type: Oscilloscope
   Manufacturer: Tektronix
   Model: TDS 210

   Driver Information
   DriverType: MATLAB Instrument Driver
   DriverName: `tektronix_tds210_ex.mdd`
   DriverVersion: 1.0

   Communication State
   Status: closed

   ```matlab
   instrhwinfo(obj)
   ans =
   Manufacturer: 'Tektronix'
   Model: 'TDS 210'
   ```
Type: 'Oscilloscope'
DriverName: 'h:\documents\tektronix_tds210_ex.mdd'

3 Connect to the instrument.
   connect(obj)

4 Verify the control commands.
   get(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 MATLAB code.

**Example: 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 code to execute on device object creation.

   ```matlab
   % Get the interface object and disconnect from instrument.
   g = get(obj, 'Interface');
   fclose(g);

   % Configure the interface object's buffers to handle up to 2500 points (two bytes per point requires 5000 bytes).
   set(g, 'InputBufferSize', 5000);
   set(g, 'OutputBufferSize', 5000);

   3. Click the **Save** button.
Verifying Create Code
This procedure verifies the MATLAB 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 Keithley 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.

   \[
g = \text{gpib('keithley', 0, 2)};
g = \text{get(g, {'InputBufferSize', 'OutputBufferSize'})}
   \]

   \[
   \begin{bmatrix}
   [512] \\
   [512]
   \end{bmatrix}
   \]

2 Create the device object, obj, using the icdevice function.

   \[
   \text{obj = icdevice('tektronix_tds210_ex.mdd', g)};
   \]

3 Verify the create code by querying the interface object’s buffer sizes.

   \[
   \text{get(g, {'InputBufferSize', 'OutputBufferSize'})}
   \]

   \[
   \begin{bmatrix}
   [5000] \\
   [5000]
   \end{bmatrix}
   \]
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 code.

**Example: 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 Keithley GPIB board at board index 0.

1. From the MATLAB command line, create the device object, `obj`, using the `icdevice` function.

   ```matlab
g = gpib('keithley', 0, 2);
obj = icdevice('tektronix_tds210_ex.mdd', g);
```

2. Connect to the instrument.

   ```matlab
   connect(obj)
   ```

3. Verify the connect code by querying the Header state of the instrument.

   ```matlab
   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 code.

**Example: 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 code to execute when the device object is disconnected from the instrument.

- Select M-Code from the Function style menu.
- Define the MATLAB code that will reset the instrument and configure the interface object’s buffers to their default values.

    % Get the interface object.
    g = get(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 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 Keithley GPIB board at board index 0. From the MATLAB command line,

1. Create the device object, `obj`, using the `icdevice` function.
   
   ```
   g = gpib('keithley', 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 14-18
“Property Components” on page 14-18
“Examples of Properties” on page 14-21

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 code.

If the set code is M-code, it can include any number of commands or MATLAB 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.

```matlab
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 code.

**Note** The M-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 string, 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 string can be restricted to accept a list of enumerated strings. For example, a property could be defined to accept the strings `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 strings `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.

<table>
<thead>
<tr>
<th>Read-Only Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Never</td>
<td>The property can be configured at all times with the set function.</td>
</tr>
<tr>
<td>While Open</td>
<td>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.</td>
</tr>
<tr>
<td>Always</td>
<td>The property cannot be configured with the set function.</td>
</tr>
</tbody>
</table>
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.

Example: 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 **General** 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 Keithley GPIB board at board index 0. From the MATLAB command line,

1. Create the device object, `obj`, using the `icdevice` function.

   ```matlab
g = gpib('keithley', 0, 2);
obj = icdevice('tektronix_tds210_ex.mdd', g);
```

2. View the `DisplayContrast` property's current value. Calling `get` on the object lists all its properties.

   ```matlab
   get(obj)
   ```

   Calling `get` on the `DisplayContrast` property lists its current value.

   ```matlab
   get(obj, 'DisplayContrast ')
   ans =
   50
   ```
3 View acceptable values for the DisplayContrast property. Calling set on the object lists all its settable properties.

    set(obj)

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.

    set(obj, 'DisplayContrast', 17)
    get(obj, 'DisplayContrast')
    ans =
        17
    set(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 General tabs.

    info = propinfo(obj,'DisplayContrast')
    info =
        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 MATLAB, the `tektronix_tds210_ex.mdd` driver actually sends the `DISplay:CONTRast?` command to the instrument.

```matlab
get(obj, 'DisplayContrast')
anst =
    17
```

When you issue the `set` function in MATLAB, the `tektronix_tds210_ex.mdd` driver actually sends the `DISplay:CONTRast` command to the instrument.

```matlab
set(obj, 'DisplayContrast', 34)
```

8 Finally, disconnect from the instrument and delete the objects.

```matlab
disconnect(obj)
delete([obj g])
```

**Example: 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.
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 **General** 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 Keithley GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

   ```matlab
   g = gpib('keithley', 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.

   ```matlab
   get(obj) 
   ```

   Calling `get` on the `CursorType` property lists its current value.

   ```matlab
   get(obj, 'CursorType') 
   ans = 
      none 
   ```

3 View acceptable values for the `CursorType` property. Calling `set` on the object lists all its settable properties.

   ```matlab
   set(obj) 
   ```

   Calling `set` on the `CursorType` property lists the values to which you can set the property.

   ```matlab
   set(obj, 'CursorType') 
   [ {none} | voltage | time ] 
   ```

4 Try setting the property to valid and invalid values.

   ```matlab
   set(obj, 'CursorType', 'voltage') 
   get(obj, 'CursorType') 
   ans = 
      voltage 
   set(obj, 'CursorType', 'horizontal') 
   ```
??? The 'horizontal' enumerated value is invalid.

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 **General** tabs.

    info = propinfo(obj,'CursorType')
info =
        Type: 'string'
        Constraint: 'enum'
        ConstraintValue: {3x1 cell}
        DefaultValue: 'none'
        ReadOnly: 'never'
        InterfaceSpecific: 1

    info.ContraintValue
ans =
    'none'
    'voltage'
    'time'

7 Connect to your instrument to verify the set and get code.

    connect(obj)

When you issue the set function in MATLAB, the `tektronix_tds210_ex.mdd` driver actually sends the CURSor:FUNCtion VBARs command to the instrument.

    set(obj, 'CursorType', 'time')

When you issue the get function in MATLAB, the `tektronix_tds210_ex.mdd` driver actually sends the CURSor:FUNCtion? command to the instrument.

    get(obj,'CursorType')
ans =
time

8 Finally disconnect from the instrument and delete the objects.

    disconnect(obj)
    delete([obj g])

Example: An M-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 code will be added to the MATLAB Set Function text field.
- The following MATLAB code is added to the MATLAB Get Function text field.

```matlab
% Extract the interface object.
interface = get(obj, 'Interface');

% Determine the type of cursor being displayed.
type = get(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 **General** 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 Keithley GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

    ```matlab
g = gpib('keithley', 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.

    ```matlab
    get(obj)
    ```

3 Calling `get` on the **CursorDelta** property lists its current value.

    ```matlab
    get(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.

    ```matlab
    set(obj)
    ```

5 Calling `set` on the **CursorDelta** property results in an error message.
set(obj, 'CursorDelta')
??? Attempt to modify read-only device property: 'CursorDelta'.

6 Setting the property to a value results in an error message.

set(obj, 'CursorDelta', 4)
??? Changing the 'CursorDelta' property of device objects is not allowed.

7 View the help you wrote.

instrhelp(obj,'CursorDelta')
CURSORDELTA (double) (read only)
Returns the difference between the two cursors.

8 List the CursorDelta characteristics that you defined in the Property Values and General tabs.

info = propinfo(obj,'CursorDelta')
info =
    Type: 'double'
    Constraint: 'none'
    ConstraintValue: []
    DefaultValue: 0
    ReadOnly: 'always'
    InterfaceSpecific: 1

9 Connect to your instrument to verify the get code.

connect(obj)

When you issue the get function in MATLAB, the tektronix_tds210_ex.mdd driver actually executes the MATLAB code that was specified.

get(obj, 'CursorDelta')
ans =
    1.6000
Finally, disconnect from the instrument and delete the objects.

```plaintext
disconnect(obj)
delete([obj g])
```
Functions

In this section...

“Understanding Functions” on page 14-34
“Function Components” on page 14-34
“Examples of Functions” on page 14-35

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 MATLAB 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.

Code
The 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 MATLAB 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 MATLAB 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 M-code used for your function’s MATLAB 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.

**Example: A 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 **Define 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 Keithley GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, `obj`, using the `icdevice` function.

   ```matlab
g = gpib('keithley', 0, 2);
obj = icdevice('tektronix_tds210_ex.mdd', g);
```
2 View the method you created.

methods(obj)

Methods for class icdevice:

<table>
<thead>
<tr>
<th>Contents</th>
<th>error</th>
<th>instrhwninfo</th>
<th>open</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>fieldnames</td>
<td>instrnotify</td>
<td>openvar</td>
</tr>
<tr>
<td>close</td>
<td>get</td>
<td>instrument</td>
<td>propinfo</td>
</tr>
<tr>
<td>connect</td>
<td>geterror</td>
<td>invoke</td>
<td>selftest</td>
</tr>
<tr>
<td>ctranspose</td>
<td>horzcat</td>
<td>isa</td>
<td>set</td>
</tr>
<tr>
<td>delete</td>
<td>icdevice</td>
<td>isequal</td>
<td>sim</td>
</tr>
<tr>
<td>devicereset</td>
<td>igetfield</td>
<td>isetfield</td>
<td>size</td>
</tr>
<tr>
<td>disconnect</td>
<td>inspect</td>
<td>isvalid</td>
<td>subsasgn</td>
</tr>
<tr>
<td>disp</td>
<td>instrcallback</td>
<td>length</td>
<td>subsref</td>
</tr>
<tr>
<td>display</td>
<td>instrfind</td>
<td>methods</td>
<td>vertcat</td>
</tr>
<tr>
<td>end</td>
<td>instrfindall</td>
<td>ne</td>
<td></td>
</tr>
<tr>
<td>eq</td>
<td>instrhelp</td>
<td>obj2mfile</td>
<td></td>
</tr>
</tbody>
</table>

Driver specific methods for class icdevice:

autoreset

3 View the help you wrote.

instrhelp(obj, 'autoreset')

INVOKE(OBJ, 'autoreset') 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, 'autoreset')
6 Disconnect from your instrument and delete the object.

```matlab
disconnect(obj)
delete([obj g])
```

**Example: A 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 New function text field and click the Add button. The new function’s name, `configureWaveform`, appears in the Function Name table.

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 New 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 \[\text{[SOURCE}, \text{START}, \text{STOP}] = \text{INVOKE(OBJ, \text{'configureWaveform'}, SOURCE, START, STOP)}\] 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 Keithley GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, obj, using the icdevice function.
g = gpib('keithley', 0, 2);
obj = icdevice('tektronix_tds210_ex.mdd', g);

2 View the method you created.

methods(obj)

Methods for class icdevice:

Contents     error     instrhwinfo     open
class         fieldnames  instrnotify    openvar
close         get         instrument   propinfo
connect       geterror    invoke       selftest
ctranspose    horzcat    isa           set
delete        icdevice    isequal      sim
devicereset    igetfield   isetfield    size
disconnect    inspect     isvalid      subsasgn
disp          instrcallback length      subsref
display       instrfind   methods      vertcat
devicereset    instrfindall ne
end            instrhelp   obj2mfile

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])

**Example: An M-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.

```matlab
function yout = getWaveform(obj)
```

• Add the following MATLAB 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.)

```matlab
% Get the interface object.
g = get(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 Keithley GPIB board at board index 0. From the MATLAB command line,

1. Create the device object, `obj`, using the `icdevice` function.

   ```matlab
   g = gpib('keithley', 0, 2);
   obj = icdevice('tektronix_tds210_ex.mdd', g);
   ```

2. View the method you created.

   ```matlab
   methods(obj)
   ```

   Methods for class `icdevice`:

   ```
   Contents   error       instrhwinfo  open
   class      fieldnames instrnotify openvar
   close      get         instrument propinfo
   connect    geterror   invoke      selftest
   ctranspose horzcat    isa         set
   delete     icdevice   isequal     sim
   devicerest reset      isetfield   size
   disconnect inspect    isvalid     subsasgn
   disp       instrcallback length   subsref
   display    instrfind   methods    vertcat
   end        instrfindall ne
   eq         instrhelp   obj2mfile
   ```

   Driver specific methods for class `icdevice`:

   ```matlab
   autoset configureWaveform getWaveform
   ```

3. View the help you wrote.

   ```matlab
   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 14-46
“Examples of Groups” on page 14-47

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.

**Example: 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 New 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.
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.`

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 Keithley GPIB board at board index 0. From the MATLAB command line,

1. Create the device object, `obj`, using the `icdevice` function.
   ```matlab
   g = gpib('keithley', 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.
   ```matlab
   get(obj)
   get(obj, 'Trigger')
   HwIndex: HwName: Type: Name:
   1 Trigger1 scope-trigger Trigger1
   ```

3. View the help.
TRIGGER (object) (read only)
Trigger is a trigger group. The trigger group object contains properties that configure and query the oscilloscope's triggering capabilities.

4 Delete the objects.

dele([obj g])

**Example: 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 General 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 **General** 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 Keithley GPIB board at board index 0. From the MATLAB command line,

1 Create the device object, obj, using the icdevice function.
g = gpib('keithley', 0, 2);
obj = icdevice('tektronix_tds210_ex.mdd', g);

2 Extract the trigger group objects, t, from the device object.

    t = get(obj, 'Trigger')
    HwIndex: HwName: Type: Name:
    1 Trigger1 scope-trigger Trigger1

3 View the current values for the properties of the trigger group object. Calling get on the object lists all its properties.

    get(t)

4 Calling get on a specific property lists its current value.

    get(t, {'Source', 'Slope'})
    ans =
        'CH1'   'falling'

5 View the acceptable values for the properties of the group object. Calling set on the object lists all its settable properties.

    set(t)

6 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 ]

7 Try setting the property to valid and invalid values.

    set(t, 'Source', 'CH2', 'Slope', 'rising')
    get(t, {'Source', 'Slope'})
    ans =
        'CH2' 'rising'
    set(t, 'Source', 'CH3')
    ??? The 'CH3' enumerated value is invalid.
set(t, 'Slope', 'steady')
??? The 'steady' enumerated value is invalid.

8 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.

9 List the group object characteristics that you defined in the Property Values and General tabs.

propinfo(t, 'Source')
ans =
    Type: 'string'
    Constraint: 'enum'
    ConstraintValue: {2x1 cell}
    DefaultValue: 'CH1'
    ReadOnly: 'never'
    InterfaceSpecific: 1

propinfo(t, 'Slope')
ans =
    Type: 'string'
    Constraint: 'enum'
    ConstraintValue: {2x1 cell}
    DefaultValue: 'falling'
    ReadOnly: 'never'
    InterfaceSpecific: 1

10 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.

```matlab
get(t, 'Source')
an =
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.

```matlab
set(t, 'Slope', 'rising')
```

11 Disconnect from your instrument and delete the objects.

```matlab
disconnect(obj)
delete([obj g])
```

**Example: 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 Keithley GPIB board at board index 0. From the MATLAB command line,
1 Create the device object, obj, using the icdevice function.

```matlab
  g = gpib('keithley', 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).

```matlab
  get(obj, 'Measurement')
```

<table>
<thead>
<tr>
<th>HwIndex</th>
<th>HwName</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Meas1</td>
<td>scope-measurement</td>
<td>Measurement1</td>
</tr>
<tr>
<td>2</td>
<td>Meas2</td>
<td>scope-measurement</td>
<td>Measurement2</td>
</tr>
<tr>
<td>3</td>
<td>Meas3</td>
<td>scope-measurement</td>
<td>Measurement3</td>
</tr>
<tr>
<td>4</td>
<td>Meas4</td>
<td>scope-measurement</td>
<td>Measurement4</td>
</tr>
</tbody>
</table>

3 View the help.

```matlab
  instrhelp(obj, 'Measurement')
```

**MEASUREMENT (object) (read only)**
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.

```matlab
  delete([obj g])
```

**Example: 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 General 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, enterSpecifies 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 **General** 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 **General** 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 \textit{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 **General** 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 \textit{Returns the measurement value}.

5 Click the **Save** button.
Verifying the Properties of the Group Object in MATLAB. 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 Keithley GPIB board at board index 0. From the MATLAB command line,

1. Create the device object, `obj`, using the `icdevice` function.

```
g = gpib('keithley', 0, 2);
obj = icdevice('tektronix_tds210_ex.mdd', g);
```

2. Extract the measurement group objects, `m`, from the device object.

```
m = get(obj, 'Measurement')
```

<table>
<thead>
<tr>
<th>HwIndex:</th>
<th>HwName:</th>
<th>Type:</th>
<th>Name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Meas1</td>
<td>scope-measurement</td>
<td>Measurement1</td>
</tr>
<tr>
<td>2</td>
<td>Meas2</td>
<td>scope-measurement</td>
<td>Measurement2</td>
</tr>
<tr>
<td>3</td>
<td>Meas3</td>
<td>scope-measurement</td>
<td>Measurement3</td>
</tr>
<tr>
<td>4</td>
<td>Meas4</td>
<td>scope-measurement</td>
<td>Measurement4</td>
</tr>
</tbody>
</table>

3. View the current values for the properties of the first group object. Calling `get` on the object lists all its properties.

```
get(m(1))
```

4. Calling `get` on a specific property lists its current value.

```
get(m(1), {'MeasurementType', 'Source', 'Units', 'Value'})
```

```
an =
'none' 'CH1' 'volts' [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))
```

```
set(m(1), 'MeasurementType')
[ frequency | period | {none} | mean | pk2pk | rms | riseTime | fallTime | posWidth | negWidth ]
```
set(m(1), 'Source', 'CH2')
ans = CH2
set(m(1), 'Source', 'CH5')
??? The 'CH5' enumerated value is invalid.

View the help you wrote.

instrhelp(m(1), 'Value')
VALUE (double) (read only)
Returns the measurement value.

List the group object characteristics that you defined in the Property Values and General tabs.

propinfo(m(1), 'Units')
ans =
  Type: 'string'
  Constraint: 'none'
  ConstraintValue: []
  DefaultValue: 'volts'
  ReadOnly: 'always'
  InterfaceSpecific: 1

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.
ans =
frequency

**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.

```matlab
set(m(2), 'Source', 'CH2')
```

**10** Disconnect from your instrument and delete the objects.

```matlab
disconnect(obj)
delete([obj g])
```
Using Existing Drivers

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Modifying MATLAB Instrument Drivers” on page 14-65</td>
</tr>
<tr>
<td>“Importing VXI plug&amp;play and IVI Drivers” on page 14-66</td>
</tr>
</tbody>
</table>

**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 14-5 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 VXIplug&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 VXIplug&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 VXIplug&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 M-code around instrument commands for analysis
• Add create, connect, and disconnect code
• Save the driver as a separate MATLAB VXIplug&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.


Setting Up Your Test (p. 15-5)  Providing global information for your test, including name, description, driver, interface, and preferences.

Defining Test Steps (p. 15-11)  Creating steps to test setting property values, getting property values, and driver functions.

Saving Your Test (p. 15-25)  Saving your test for future midtest sessions, or as M-code, or as a driver function.

Testing and Results (p. 15-28)  Running a complete or partial test; exporting or saving test results.
Instrument Driver Testing Tool: Overview

In this section...

| “Functionality” on page 15-2 |
| “Drivers” on page 15-2 |
| “Test Structure” on page 15-3 |
| “Starting” on page 15-3 |
| “Example” on page 15-4 |

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, MATLAB code, or driver function.
- Export the test results to MATLAB workspace, figure window, MAT-file, or the MATLAB array 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 VXIplug&play drivers
• MATLAB IVI drivers

MATLAB VXIplug&play drivers and MATLAB IVI drivers can be created from VXIplug&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')

Example

For the examples in this chapter, you will create a test for the Tektronix TDS210 oscilloscope driver that you created in Chapter 14, “Using the Instrument Driver Editor”.

You will create each kind of step in your test: set property, get property, sweep properties, and function.
Setting Up Your Test

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.

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, TCP/IP, 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.
Example: 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 Chapter 14, “Using the Instrument Driver Editor”.

1. Open the MATLAB Instrument Driver Testing Tool from the command line with the command `mtest`.

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 Chapter 14, “Using the Instrument Driver Editor”. 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,

- Select your **Instrument object type**, **Vendor**, **Board index**, and **Primary address** of your instrument.

  The example illustrations in this chapter use a CEC GPIB board with index 0 and the instrument at address 4. Your configuration may be different.

- 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 15-11
“Test Step: Get Property” on page 15-15
“Test Step: Properties Sweep” on page 15-17
“Test Step: Function” on page 15-21

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.

<table>
<thead>
<tr>
<th>Settings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>You provide a name for each test step. The name appears in the Test Explorer tree as well as in the results output.</td>
</tr>
<tr>
<td>Property to Test</td>
<td>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.</td>
</tr>
<tr>
<td>Object(s) to Test</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Settings Description

<table>
<thead>
<tr>
<th>Settings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Define the Values to Test</td>
<td>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 15-6).</td>
</tr>
</tbody>
</table>

### 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.

### Example: 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 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 15-11, 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 15-30, after the test step has executed.

**Example: 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 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 15-6, 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.

**Example: 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 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, 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 15-30 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 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.
Example: 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 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.
### Defining Test Steps

#### MATLAB Instrument Driver Testing Tool

**Test Explorer**
- IDS 210 Driver Sample Test
  - Set Display Contrast
  - Get Display Contrast
  - All Properties Sweep
  - Config Platform

**Config Waveform (Function Test Step)**

<table>
<thead>
<tr>
<th>Name:</th>
<th>Config waveform</th>
</tr>
</thead>
</table>

**Function to test:**

- Device object functions:
  - configure waveform
  - device reset

**Function definition:**

```
[OUT1, OUT2, OUT3] = INVOKEOBJ('configureWaveform', 'SOURCE', ST. Object(s) to test:

- Device object
- Input argument(s):
  - OFF, 1, 3000

**Output argument(s):**

- Channel, StartAdv, StopAdv

Select when this step passes:

- If no MATLAB or instrument error occurs
- If output arguments are OFF, 1, 2500
- If output of function: OFF, 1, 2500 is true

[Show Help on Function...]
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.
Saving Your Test

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Saving the Test as M-Code” on page 15-25</td>
</tr>
<tr>
<td>“Saving the Test as a Driver Function” on page 15-25</td>
</tr>
</tbody>
</table>

Saving the Test as M-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 M-code by clicking the **File** menu and selecting **Save Test as M-Code**.

You can execute the test by calling this M-file from the MATLAB command line.

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 line 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 line or in an M-file.

**Example: 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.

**Example: Calling a Driver Test Function from MATLAB**

Now that the test function is included in the driver, you access it with the invoke command from the MATLAB command line.

In the MATLAB Command Window,

1. Create an interface object.
   
   ```matlab
   g = gpib('cec',0,4)
   ```

   2. Create a device object, specifying the driver with the **drivertest** function saved in it.
   
   ```matlab
   obj = icdevice('tektronix_tds210_ex.mdd',g)
   ```

3. Connect to the device.
   
   ```matlab
   connect(obj)
   ```

4. Execute the driver test.
   
   ```matlab
   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

<table>
<thead>
<tr>
<th>In this section...</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Running All Steps” on page 15-28</td>
</tr>
<tr>
<td>“Partial Testing” on page 15-30</td>
</tr>
<tr>
<td>“Exporting Results” on page 15-30</td>
</tr>
<tr>
<td>“Saving Results” on page 15-31</td>
</tr>
</tbody>
</table>

**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.

**Example: 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.
Testing and Results

MATLAB Instrument Driver Testing Tool

Test Explorer:
- TDS 210 Driver Sample Test
  - Set Display Contrast
  - Set Display Contrast
  - All Properties Sweep
  - Config Waveform

Output Window:
- Test: TDS 210 Driver Sample Test
  - Start time: 23-Dec-2003 14:51:44

Test Step 1: Set Display Contrast (Set Property Test Step)
- Test step status: PASSED

Test Step 2: Get Display Contrast (Get Property Test Step)
- Test step status: PASSED

Test Step 3: All Properties Sweep (Properties Sweep Test Step)
- Testing CursorDelta property in the parent group.
- Testing CursorType property in the parent group.
- Testing DisplayContrast property in the parent group.
- Testing InstrumentModel property in the parent group.
- Testing MeasurementType property in the Measurement group.
- Testing Source property in the Measurement group.
- Testing Units property in the Measurement group.
- Testing Value property in the Measurement group.
- Testing Slope property in the Trigger group.
- Testing Source property in the Trigger group.
- Test step status: PASSED

Test Step 4: Config Waveform (Function Test Step)
- Test step status: PASSED

Step time: 23-Dec-2003 14:52:13
Test status: PASSED
**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 Array Editor

The results you can export are those assigned to output variables in the settings for a test step.

**Example: 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.

![Testing and Results](image)

**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**.
Using the Instrument Control Toolbox Block Library

Instrument Control Toolbox includes a Simulink® 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.

- **Overview (p. 16-2)**
  - Introduces the Instrument Control Toolbox block library

- **Opening the Instrument Control Block Library (p. 16-3)**
  - Describes how to access the block library

- **Building Simulink Models to Send and Receive Data (p. 16-6)**
  - Examples of using Instrument Control blocks.
Overview

This chapter describes how to use the Instrument Control Toolbox block library. The Instrument Control Toolbox block library consists of six blocks:

- **Query Instrument** — Query the instrument for data.
- **To Instrument** — Send data to the instrument.
- **TCPIP Receive** — Receive data over a TCP/IP network.
- **TCPIP Send** — Send data over TCP/IP network.
- **UDP Receive** — Receive data over UDP network.
- **UDP Send** — Send data over UDP network.

The Instrument Control Toolbox block library is a tool for sending live data from your model to an instrument, or querying an instrument to receive live data into your model. You can use blocks from the block library with blocks from other Simulink libraries to create sophisticated models.

To use the Instrument Control Toolbox block library you require Simulink, a tool for simulating dynamic systems. Simulink is a model definition environment. Use Simulink blocks to create a block diagram that represents the computations of your system or application. Simulink is also a model simulation environment. Run the block diagram to see how your system behaves. If you are new to Simulink, read *Getting Started with Simulink* in the Simulink documentation to better understand its functionality.

For more detailed information about the blocks in the Instrument Control Toolbox block library, see Blocks — Alphabetical List. For example for using the Instrument Control Toolbox block library, see “Building Simulink Models to Send and Receive Data” on page 16-6.
Opening the Instrument Control Block Library

In this section...

- “Using the instrumentlib Command from MATLAB” on page 16-3
- “Using the Simulink Library Browser” on page 16-4

Using the instrumentlib Command from MATLAB

To open the Instrument Control block library, enter

    instrumentlib

at the MATLAB prompt. MATLAB displays the contents of the library in a separate window.
Using the Simulink Library Browser

To open the Instrument Control Toolbox block library, start the Simulink Library Browser and select the library from the list of available block libraries displayed in the browser.

To start the Simulink Library Browser, enter

```
simulink
```

at the MATLAB prompt. MATLAB opens the browser window. The left pane lists available block libraries, with the basic Simulink library listed first, followed by other libraries listed in alphabetical order under it. To open the Instrument Control Toolbox block library, click its icon.
Simulink loads the library and displays the blocks in the library.
Building Simulink Models to Send and Receive Data

In this section...

| “Example: Sending and Receiving Data Through a Serial Port Loopback” on page 16-6 |
| “Example: Sending and Receiving Data Via a TCP/IP Network” on page 16-16 |

Example: Sending and Receiving Data Through a Serial Port Loopback

This section provides an example that builds a simple model using the Instrument Control 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 COM1 serial port and to read that data back into your model.

You will 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.

- “Step 1: Open the Block Library” on page 16-6
- “Step 2: Create a New Model” on page 16-7
- “Step 3: Drag the Instrument Control Toolbox Blocks into the Model” on page 16-7
- “Step 4: Drag Other Blocks to Complete the Model” on page 16-8
- “Step 5: Connect the Blocks” on page 16-10
- “Step 6: Specify the Block Parameter Values” on page 16-11
- “Step 7: Run the Simulation” on page 16-14

Step 1: Open the Block Library

To open the Instrument Control Toolbox block library, start the Simulink Library Browser and choose Instrument Control Toolbox from the list of available libraries displayed in the browser.
To start the Simulink Library Browser, enter

```
simulink
```

at the MATLAB prompt. In the **Simulink Library Browser** the left pane lists the available block libraries. To open the Instrument Control Toolbox block library, click its entry icon. See “Using the Simulink Library Browser” on page 16-4 for more information.

**Step 2: Create a New Model**

To use a block, you must add it to an existing model or create a new model.

For this example create a new model by clicking the **Create a new model** button on the toolbar.

You can also click the **File** menu in the Simulink Library Browser and select **New > Model**. Simulink opens an empty model window on the display. To assign the new model a name, use the **Save** option.

**Step 3: Drag the Instrument Control Toolbox Blocks into the Model**

To use the blocks in a model, click a block in the library and, holding the mouse button down, drag it into the model window. For this example, you will need one instance of the To Instrument and the Query Instrument block in your model.
Drag Instrument Control Toolbox Blocks into Model Window

**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 will be a constant, you can use the Constant block for this purpose. To access the Constant block, expand the Simulink node in the browser tree, and click on the Source library entry. From
the blocks displayed in the right pane, drag the Constant block into the model and place it to the left of the To Instrument block.

Drag Constant Block to Model Window

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 model and place it to the right of the Query Instrument block.
Drag Display Block to Model Window

**Step 5: Connect the Blocks**

Make a connection between the Constant block and the To Instrument block. When you move the pointer near the output port of the Constant block, the pointer become a cross hair. Click on the Constant output port and holding the mouse button, drag the pointer to the input port of the To Instrument block, and release the button.

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 together within the model. The only communication between them is through the instrument, which is the loopback connected to the COM1 serial port. Because there is no direct connection between these two blocks, 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
You set parameters for the blocks in your model by double-clicking on the block.

Configure the To Instrument Block. Double-click the To Instrument block to open its parameters dialog box. Set the block sample time to 0.8. In the Hardware Configuration tab, specify new hardware configuration with the values shown in the following figure.
Using the Instrument Control Toolbox Block Library

The model uses the default values on the **Instrument Initialization** and **Send** tabs of this block, so you do not need to modify any of their values.

Click **Apply**, then **OK**.

**Configure the Query Instrument Block.** Double-click the Query Instrument block to open its parameters dialog box. Set the block sample time to 1. Note that this time is slightly longer than the setting for the To Instrument block. This is to assure that the To Instrument is complete in writing data to the loopback before the Query Instrument block reads it.

In the **Hardware Configuration** tab, specify new a hardware configuration with values to match the settings in the To Instrument block as shown in the following figure.
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 **Apply**, then **OK**.

**Configure the Constant Block.** Double-click the Constant block to open its parameters dialog box. Change the Constant value to whatever value you want to send to the instrument. This example uses 25, as shown. You can leave other setting in this dialog box at their default values.
Click **Apply**, then **OK**.

**Configure the Display Block.** For the Display block, you can use its default parameters.

**Step 7: Run the Simulation**

To run the simulation, click the Start button on the model window toolbar. Alternatively, you can use the **Simulation** menu in the model window and choose the **Start** option.

When you run the simulation, the constant value you specified (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 model window updates the progress of the simulation.
Example: Sending and Receiving Data Via a TCP/IP Network

This section provides an example that builds a simple model using the Instrument Control blocks in the block library in conjunction with other blocks in the Simulink library. This example illustrates how to send data to an echo server using TCP/IP and to read that data back into your model.

You will create an echo server on your machine that will simulate sending a signal to the TCP/IP send block and echo the result back to the Send block to send data, and then use the TCP/IP Receive block to read that same data back into your model.

- “Step 1: Create an Echo Server” on page 16-16
- “Step 2: Open the Block Library” on page 16-17
- “Step 3: Create a New Model” on page 16-17
- “Step 4: Drag the Instrument Control Toolbox Blocks into the Model” on page 16-17
- “Step 5: Drag Other Blocks to Complete the Model” on page 16-18
- “Step 6: Connect the Blocks” on page 16-20
- “Step 7: Specify the Block Parameter Values” on page 16-21
- “Step 8: Specify the Block Priorities” on page 16-23
- “Step 9: Run the Simulation” on page 16-26
- “Step 10: View the Result” on page 16-27

Step 1: Create an Echo Server

You should first open a port on your computer that will work as an echo server that you can use to send and receive signals via TCP/IP. To create an echo server type the following command in MATLAB:

```matlab
echotcpip('on', 50000)
```

This will open port 50000 on your machine to work as an echo server and turn it on.
**Step 2: Open the Block Library**

To open the Instrument Control Toolbox block library, start the Simulink Library Browser and choose Instrument Control Toolbox from the list of available libraries displayed in the browser.

To start the Simulink Library Browser, enter

```
simulink
```

at the MATLAB prompt. In the *Simulink Library Browser* the left pane lists the available block libraries. To open the Instrument Control Toolbox block library, click its entry icon. See “Using the Simulink Library Browser” on page 16-4 for more information.

**Step 3: Create a New Model**

To use a block, you must add it to an existing model or create a new model.

For this example, create a new model by clicking the *Create a new model* button on the toolbar.

You can also click the **File** menu in the Simulink Library Browser and select **New > Model**. Simulink opens an empty model window on the display. To assign the new model a name, use the **Save** option.

**Step 4: Drag the Instrument Control Toolbox Blocks into the Model**

To use the blocks in a model, click each block in the library and, holding the mouse button down, drag it into the model window. For this model, you will need one instance of the TCP/IP Send and the TCP/IP Receive block in your model.
Drag Instrument Control Toolbox Blocks into Model Window

**Step 5: Drag Other Blocks to Complete the Model**

This example requires two more blocks. One block displays the data received from the receive block and the other block is the data to be sent to the 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 on the Sources library entry. From the blocks displayed in the right pane, drag the Sine Wave block into the model and place it to the left of the TCP/IP Send block.

Drag Sine Wave Block to Model Window

To display the data received by the TCP/IP Receive block, you can use the Scope block. To access the Scope block, click the Siping nks 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 model and place it to the right of the Query Instrument block.

Drag Scope Block to Model Window

**Step 6: Connect the Blocks**
Make a connection between the Sine Wave block and the TCP/IP Send block. When you move the pointer near the output port of the Sine Wave block, the pointer becomes a cross hair. Click on the Sine Wave output port and holding
the mouse button, drag the pointer to the input port of the TCP/IP Send block, and release the button. 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.

The two Instrument Control Toolbox blocks do not directly connect together within the model. You should therefore, set the block **Priority** to specify the order of execution of the blocks. Without the priorities specified, there is a chance that, during simulation the Receive block can run before the Send block. See “Step 8: Specify the Block Priorities” on page 16-23, for more information.

**Step 7: Specify the Block Parameter Values**
You set parameters for the blocks in your model by double-clicking on 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.
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` and the Block sample time to `0.1`. 
Click **OK**.

**Configure the Sine Wave Block.** Double-click on the Sine Wave to open its parameters dialog box. Set the Sample time to 0.1.

Click **OK**.

**Step 8: Specify the Block Priorities**

In order for the simulation to run correctly, you should specify the order in which Simulink should process the blocks. To do this you should set block priority from the **Block Properties** dialog box.

**Note** It is essential that you set the correct priority for the blocks in your model, otherwise your simulation will fail.

- Right-click on the **Sine Wave** block and select **Block Properties** from the pop-up menu.
In the Block Properties dialog box, enter 1 in the **Priority** field. This tells Simulink to run this block first.
Click **Apply** and then **OK**.

- Right-click on the **TCP/IP Send** block and select **Block Properties** from the pop-up menu.

  In the Block Properties dialog box, enter 2 in the **Priority** field. This tells Simulink to run the block second.

  Click **Apply** and then **OK**.

- Right-click on the **TCP/IP Receive** block and select **Block Properties** from the pop-up menu.

  In the Block Properties dialog box, enter 3 in the **Priority** field. This tells Simulink to run the block third.

  Click **Apply** and then **OK**.
• Right-click on the **Scope** block and select Block Properties from the pop-up menu.

In the Block Properties dialog box, enter 4 in the **Priority** field. This tells Simulink to run the block fourth.

Click **Apply** and then **OK**.

Read Working with Block Parameters in the Simulink User Guide to understand more about block priorities.

**Step 9: Run the Simulation**

To run the simulation, click **Start** on the model window toolbar. Alternatively, you can use the Simulation menu in the model window and choose the Start option.

While the simulation is running, the status bar at the bottom of the model window updates the progress of the simulation.
Step 10: View the Result
You can double-click the Scope block to view the signal on a graph as it is received by the TCP/IP Receive block.
# Functions — By Category

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instrument Object Creation (p. 17-2)</td>
<td>Create interface objects and device objects</td>
</tr>
<tr>
<td>State Change (p. 17-2)</td>
<td>Connect to instruments, disconnect from instruments, and record information to disk</td>
</tr>
<tr>
<td>Property Display and Configuration (p. 17-3)</td>
<td>Return properties, configure properties, and open property inspector</td>
</tr>
<tr>
<td>Reading Data (p. 17-3)</td>
<td>Read text, binary, and binblock data for interface objects</td>
</tr>
<tr>
<td>Writing Data (p. 17-4)</td>
<td>Write text, binary, and binblock data for interface objects</td>
</tr>
<tr>
<td>Information and Help (p. 17-5)</td>
<td>Return function help, property help, and property characteristics</td>
</tr>
<tr>
<td>Graphical Tools (p. 17-5)</td>
<td>Tools that facilitate instrument control in graphical environment</td>
</tr>
<tr>
<td>General Purpose (p. 17-6)</td>
<td>Delete, clear, find, save, and load objects, convert objects to M-code</td>
</tr>
<tr>
<td>Interface Objects (p. 17-7)</td>
<td>Functions listed by interface type (GPIB, VISA-VXI, and so on)</td>
</tr>
<tr>
<td>Device Objects (p. 17-10)</td>
<td>Functions used only with device objects</td>
</tr>
<tr>
<td>IVI Configuration Store Objects (p. 17-10)</td>
<td>Functions used only with IVI configuration store objects</td>
</tr>
</tbody>
</table>
**Instrument Object Creation**

- Interface Object (p. 17-2)  Create interface object
- Device Object (p. 17-2)  Create device object

**Interface Object**

- gpib  Create GPIB object
- serial  Create serial port object
- tcpip  Create TCPIP object
- udp  Create UDP object
- visa  Create VISA object

**Device Object**

- icdevice  Create device object

**State Change**

- Interface Object (p. 17-3)  Control interface object
- Device Object (p. 17-3)  Control device object
**Interface Object**

- **fclose**
  - Disconnect interface object from instrument

- **fopen**
  - Connect interface object to instrument

- **record**
  - Record data and event information to file

**Device Object**

- **connect**
  - Connect device object to instrument

- **disconnect**
  - Disconnect device object from instrument

**Property Display and Configuration**

- **get**
  - Instrument object properties

- **inspect**
  - Open Property Inspector

- **set**
  - Configure or display instrument object properties

**Reading Data**

- **binblockread**
  - Read binblock data from instrument

- **fgets**
  - Read line of text from instrument and include terminator

- **fgetl**
  - Read line of text from instrument and discard terminator
### Functions — By Category

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flushinput</td>
<td>Remove data from input buffer</td>
</tr>
<tr>
<td>fread</td>
<td>Read binary data from instrument</td>
</tr>
<tr>
<td>fscanf</td>
<td>Read data from instrument, and format as text</td>
</tr>
<tr>
<td>query</td>
<td>Write text to instrument, and read data from instrument</td>
</tr>
<tr>
<td>readasync</td>
<td>Read data asynchronously from instrument</td>
</tr>
<tr>
<td>scanstr</td>
<td>Read data from instrument, format as text, and parse</td>
</tr>
<tr>
<td>stopasync</td>
<td>Stop asynchronous read and write operations</td>
</tr>
</tbody>
</table>

### Writing Data

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binblockwrite</td>
<td>Write binblock data to instrument</td>
</tr>
<tr>
<td>flushoutput</td>
<td>Remove data from output buffer</td>
</tr>
<tr>
<td>fprintf</td>
<td>Write text to instrument</td>
</tr>
<tr>
<td>fwrite</td>
<td>Write binary data to instrument</td>
</tr>
<tr>
<td>query</td>
<td>Write text to instrument, and read data from instrument</td>
</tr>
<tr>
<td>stopasync</td>
<td>Stop asynchronous read and write operations</td>
</tr>
</tbody>
</table>
Information and Help

instrhelp  
Help for instrument object type, function, or property

instrhwinfo  
Information about available hardware

propinfo  
Instrument object property information

Graphical Tools

Interface Object (p. 17-5)  
Graphical tools for interface objects

Device Object (p. 17-5)  
Graphical tools for device objects

Interface Object

tmtool  
Open Test & Measurement Tool

Device Object

midedit  
Open graphical tool for creating and editing MATLAB instrument driver

midtest  
Open graphical tool for testing MATLAB instrument driver

tmtool  
Open Test & Measurement Tool
## General Purpose

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear</td>
<td>Remove instrument objects from MATLAB workspace</td>
</tr>
<tr>
<td>delete</td>
<td>Remove instrument objects from memory</td>
</tr>
<tr>
<td>disp</td>
<td>Display instrument object summary information</td>
</tr>
<tr>
<td>instrcallback</td>
<td>Display event information when event occurs</td>
</tr>
<tr>
<td>instrfind</td>
<td>Read instrument objects from memory to MATLAB workspace</td>
</tr>
<tr>
<td>instrfindall</td>
<td>Find visible and hidden instrument objects</td>
</tr>
<tr>
<td>instrid</td>
<td>Define and retrieve commands that identify instruments</td>
</tr>
<tr>
<td>instrnotify</td>
<td>Define notification for instrument events</td>
</tr>
<tr>
<td>instrreset</td>
<td>Disconnect and delete all instrument objects</td>
</tr>
<tr>
<td>isvalid</td>
<td>Determine whether instrument objects are valid</td>
</tr>
<tr>
<td>length</td>
<td>Length of instrument object array</td>
</tr>
<tr>
<td>load</td>
<td>Load instrument objects and variables into MATLAB workspace</td>
</tr>
<tr>
<td>methods</td>
<td>Class method names and descriptions</td>
</tr>
<tr>
<td>obj2mfile</td>
<td>Convert instrument object to MATLAB code</td>
</tr>
<tr>
<td>save</td>
<td>Save instrument objects and variables to MAT-file</td>
</tr>
<tr>
<td>size</td>
<td>Size of instrument object array</td>
</tr>
</tbody>
</table>
## Interface Objects

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIB (p. 17-7)</td>
<td>Create GPIB object, clear instrument buffer, send trigger message, perform serial poll</td>
<td></td>
</tr>
<tr>
<td>Serial Port (p. 17-8)</td>
<td>Create serial port object, send break to instrument</td>
<td></td>
</tr>
<tr>
<td>TCP/IP (p. 17-8)</td>
<td>Create TCP/IP object, return host information, start or stop echo server</td>
<td></td>
</tr>
<tr>
<td>UDP (p. 17-8)</td>
<td>Create UDP object, return host information, start or stop echo server</td>
<td></td>
</tr>
<tr>
<td>VISA-GPIB (p. 17-8)</td>
<td>Create VISA-GPIB object, clear instrument buffer, send trigger message</td>
<td></td>
</tr>
<tr>
<td>VISA-GPIB-VXI (p. 17-8)</td>
<td>Create VISA-GPIB-VXI object, low level memory operations, high level memory operations, send trigger</td>
<td></td>
</tr>
<tr>
<td>VISA-Serial (p. 17-9)</td>
<td>Create VISA-Serial object</td>
<td></td>
</tr>
<tr>
<td>VISA-VXI (p. 17-9)</td>
<td>Create VISA-VXI object, low level memory operations, high level memory operations, send trigger</td>
<td></td>
</tr>
</tbody>
</table>

### GPIB

- `clrdevice` - Clear instrument buffer
- `gpib` - Create GPIB object
- `spoll` - Perform serial poll
- `trigger` - Send trigger message to instrument
Serial Port

**serial**
Create serial port object

**serialbreak**
Send break to instrument

TCP/IP

**echotcpip**
Start or stop TCP/IP echo server

**resolvehost**
Network name or network address

**tcpip**
Create TCPIP object

UDP

**echoudp**
Start or stop UDP echo server

**resolvehost**
Network name or network address

**udp**
Create UDP object

VISA-GPIB

**clrdevice**
Clear instrument buffer

**trigger**
Send trigger message to instrument

**visa**
Create VISA object

VISA-GPIB-VXI

**clrdevice**
Clear instrument buffer

**memmap**
Map memory for low-level memory read and write operations

**mempeek**
Low-level memory read from VXI register
mempoke

Low-level memory write to VXI register

memread

High-level memory read from VXI register

memunmap

Unmap memory for low-level memory read and write operations

memwrite

High-level memory write to VXI register

visa

Create VISA object

**VISA-Serial**

visa

Create VISA object

**VISA-VXI**

clrdevice

Clear instrument buffer

memmap

Map memory for low-level memory read and write operations

mempeek

Low-level memory read from VXI register

mempoke

Low-level memory write to VXI register

memread

High-level memory read from VXI register

memunmap

Unmap memory for low-level memory read and write operations

memwrite

High-level memory write to VXI register

trigger

Send trigger message to instrument

visa

Create VISA object
### Device Objects

- **devicereset**: Reset instrument
- **geterror**: Check and return error message from instrument
- **invoke**: Execute driver-specific function on device object
- **makemid**: Convert driver to MATLAB instrument driver format
- **selftest**: Run instrument self-test

### IVI Configuration Store Objects

- **add**: Add entry to IVI configuration store object
- **commit**: Save IVI configuration store object to data file
- **iviconfigurationstore**: Create IVI configuration store object
- **remove**: Remove entry from IVI configuration store object
- **update**: Update entry of IVI configuration store object
Functions — Alphabetical List
add

**Purpose**
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'**
  Type of entry being added
- **'HardwareAsset'**
  Type of entry being added
- **'LogicalName'**
  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
```
**add**

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Any string</td>
<td>Description of driver session</td>
</tr>
<tr>
<td>VirtualNames</td>
<td>structure</td>
<td>A struct array containing virtual name mappings</td>
</tr>
<tr>
<td>Cache</td>
<td>on/off</td>
<td>Enable caching if the driver supports it.</td>
</tr>
<tr>
<td>DriverSetup</td>
<td>Any string</td>
<td>This value is software module dependent</td>
</tr>
<tr>
<td>InterchangeCheck</td>
<td>on/off</td>
<td>Enable driver interchangeability checking, if supported</td>
</tr>
<tr>
<td>QueryInstrStatus</td>
<td>on/off</td>
<td>Enable instrument status querying by the driver</td>
</tr>
<tr>
<td>RangeCheck</td>
<td>on/off</td>
<td>Enable extended range checking by the driver, if supported</td>
</tr>
<tr>
<td>RecordCoercions</td>
<td>on/off</td>
<td>Enable recording of coercions by the driver, if supported</td>
</tr>
<tr>
<td>Simulate</td>
<td>on/off</td>
<td>Enable simulation by the driver</td>
</tr>
</tbody>
</table>

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,
add

IOResourceDescriptor. Optional parameter-value pairs may be included.

Valid parameters for HardwareAsset are

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Any string</td>
<td>Description of hardware asset</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Any string</td>
<td>Description of logical name</td>
</tr>
</tbody>
</table>

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.

Examples

Construct IVI configuration store object, c.

```plaintext
c = iviconfigurationstore;
```

Add a hardware asset with name gpib1, and resource description GPIBO::1::INSTR.

```plaintext
add(c, 'HardwareAsset', 'gpib1', 'GPIBO::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::INSTR.

s.Type = 'HardwareAsset';
s.Name = 'gpib3';
s.IOResourceDescriptor = 'GPIB0::3::INSTR';
add(c, s);

Save the changes to the IVI configuration store data file.

c commit(c);

See Also

Functions

iviconfigurationstore, iviconfigurationstore/commit, iviconfigurationstore/remove, iviconfigurationstore/update
binblockread

**Purpose**
Read binblock data from instrument

**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 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.

**Remarks**

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 line 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.

**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.

```matlab
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.

```matlab
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.
binblockread

```matlab
fprintf(g,'CURVE?')
data = binblockread(g);
```

**See Also**

**Functions**

- binblockwrite
- fopen
- fread
- instrhelp

**Properties**

- BytesAvailable
- InputBufferSize
- Status
- ValuesReceived
Purpose  Write binblock data to instrument

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 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 MATLAB values to the specified precision, \textit{precision}. The ASCII header, header, is prefixed to the binblock.

binblockwrite(obj,A,'precision','header','headerformat') writes binary data, A, translating MATLAB values to the specified precision, \textit{precision}. The ASCII header, header, is prefixed to the binblock using the format specified by \textit{headerformat}.

\textit{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, f, e, E, g, G, c, and s. Type \texttt{instrhelp fprintf} for more information on valid values for \textit{headerformat}. By default, \textit{headerformat} is \%s.

**Remarks**

Before you can write data to the instrument, it must be connected to obj with the \texttt{fopen} function. A connected interface object has a \texttt{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 \texttt{ValuesSent} property value is increased by the number of values written each time \texttt{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 \texttt{OutputBufferSize} property.

**Examples**

```matlab
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);

See Also

Functions
binblockread, fopen, fwrite, instrhelp

Properties
OutputBufferSize, OutputEmptyFcn, Status, Timeout,
TransferStatus, ValuesSent
clear

**Purpose**
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.

**Remarks**
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.

**Examples**
This example creates the GPIB object g, copies g to a new variable gcop y, and clears g from the MATLAB workspace. g is then restored to the workspace with instrfind and is shown to be identical to gcop y.

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

```
g = instrfind;
isequal(gcopy,g)
an s  =
   1
```

**See Also**

**Functions**
delete, fclose, instrfind, instrhelp, isvalid

**Properties**

Status
Purpose
Clear instrument buffer

Syntax
clrdevice(obj)

Arguments
obj A GPIB, VISA-GPIB, VISA-VXI, or VISA-GPIB-VXI object.

Description
clrdevice(obj) clears the hardware buffer of the instrument connected to obj.

Remarks
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.

See Also
Functions
flushinput, flushoutput, fopen

Properties
Status
Purpose
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
Functions
iviconfigurationstore, iviconfigurationstore/add,
iviconfigurationstore/remove, iviconfigurationstore/update
**Purpose**
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`.

**Remarks**
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.

**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(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)
dele([d g])
```

**See Also**

**Functions**
disconnect, delete, instrhelp

**Properties**
Status
### Purpose
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.

### Remarks
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.

### 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`.

```matlab
  g = gpib('ni',0,1);
  fopen(g)
  fprintf(g,'*IDN?')
  idn = fscanf(g);
  fclose(g)
  delete(g)
  clear g
```
See Also

Functions

clear, fclose, instrhelp, isValid, stopasync

Properties

Status
**Purpose**  
Reset instrument

**Syntax**  
`devicereset(obj)`

**Arguments**  
`obj`  
A device object.

**Description**  
`devicereset(obj)` resets the instrument associated with the device object specified by `obj`. 
disconnect

**Purpose**
 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.

**Remarks**
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.

**Examples**
Create a device object for a Tektronix TDS 210 oscilloscope that is connected to a National Instruments GPIB board.

```matlab
    g = gpib('ni',0,2);
    d = icdevice('tektronix_tds210',g);
    connect(d)
    values = get(d);
    disconnect(d)
    delete([d g])
```
See Also

Functions
connect, delete, instrhelp

Properties
Status
**disp**

**Purpose**
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`.

**Remarks**

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.

**Examples**

The following commands display summary information for the GPIB object `g`.

```
g = gpib('ni',0,1)
g.EOSMode = 'read'
g
```
**Purpose**  
Start or stop TCP/IP echo server

**Syntax**  
echotcpip('state',port)  
echotcpip('state')

**Arguments**  
'state'  
Turn the server on or off.  
port  
Port number of the server.

**Description**  
echotcpip('state',port) starts a TCP/IP server with port number specified by port. state can only be on.  
echotcpip('state') stops the echo server. state can only be off.

**Examples**  
Start the echo server and create a TCPIP object.

```matlab
    echotcpip('on',4000)  
    t = tcpip('localhost',4000);
```

Connect the TCPIP object to the host.

```matlab
    fopen(t)
```

Write to the host and read from the host.

```matlab
    fprintf(t,'echo this string.')  
    data = fscanf(t);
```

Stop the echo server and disconnect the TCPIP object from the host.

```matlab
    echotcpip('off')  
    fclose(t)
```

**See Also**  
**Functions**  
echoudp, tcpip, udp
**echoudp**

**Purpose**
Start or stop UDP echo server

**Syntax**
echoudp('state', port)
echoudp('state')

**Arguments**
- `'state'` Turn the server on or off.
- `port` Port number of the server.

**Description**
echoudp('state', `port`) starts a UDP server with port number specified by `port`. `state` can only be `on`.
echoudp('state') stops the echo server. `state` can only be `off`.

**Examples**
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 read from the host.

```
fwrite(u,65:74)
A = fread(u,10);
```

Stop the echo server and disconnect the UDP object from the host.

```
echoudp('off')
fclose(u)
```

**See Also**
Functions
echotcpip, tcpip, udp
**Purpose**
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.

**Remarks**
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.

**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.

```plaintext
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.

**See Also**

**Functions**
clear, delete, fopen, instrhelp, record, stopasync
Properties

RecordStatus, Status
Purpose
Read line of text from instrument and discard terminator

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.

Remarks
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.

**Rules for Completing a Read Operation with `fgetl`**

A read operation with `fgetl` blocks access to the MATLAB command line until

- The terminator is read. For serial port, TCP/IP, 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.

**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.
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.

```matlab
        g = gpib('ni',0,1);
fopen(g)
g.EOSMode = 'read';
fprintf(g,'*IDN?')
```

Asynchronously read the identification information from the instrument.

```matlab
readasync(g)
g.BytesAvailable
ans =
      56
```

Use fgetl to transfer the data from the input buffer to the MATLAB workspace, and discard the terminator.

```matlab
idn = fgetl(g)
idn =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

```matlab
length(idn)
an =
      55
```

Disconnect g from the scope, and remove g from memory and the workspace.

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

See Also

Functions

fgets, fopen, instrhelp
Properties
BytesAvailable, EOSCharCode, EOSMode, InputBufferSize, Status, Terminator, Timeout, ValuesReceived
Purpose
Read line of text from instrument and include terminator

Syntax
\[ \text{tline} = \text{fgets}(\text{obj}) \]
\[ [\text{tline}, \text{count}] = \text{fgets}(\text{obj}) \]
\[ [\text{tline}, \text{count}, \text{msg}] = \text{fgets}(\text{obj}) \]
\[ [\text{tline}, \text{count}, \text{msg}, \text{datagramaddress}, \text{datagramport}] = \text{fgets}(\text{obj}) \]

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
\( \text{tline} = \text{fgets}(\text{obj}) \) reads one line of text from the instrument connected to \text{obj}, and returns the data to \text{tline}. The returned data includes the terminator with the text line. To exclude the terminator, use \text{fgetl}.

\[ [\text{tline}, \text{count}] = \text{fgets}(\text{obj}) \] returns the number of values read to \text{count}.

\[ [\text{tline}, \text{count}, \text{msg}] = \text{fgets}(\text{obj}) \] returns a warning message to \text{msg} if the read operation was unsuccessful.

\[ [\text{tline}, \text{count}, \text{msg}, \text{datagramaddress}, \text{datagramport}] = \text{fgets}(\text{obj}) \] returns the remote address and port from which the datagram originated. These values are returned only if \text{obj} is a UDP object.

Remarks
Before you can read text from the instrument, it must be connected to \text{obj} with the \text{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.

**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, TCP/IP, 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.

**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.
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.

```matlab
g = gpib('ni',0,1);
fopen(g)
g.EOSMode = 'read';
fprintf(g,'*IDN?')
```

Asynchronously read the identification information from the instrument.

```matlab
readasync(g)
g.BytesAvailable
ans =
  56
```

Use `fgets` to transfer the data from the input buffer to the MATLAB workspace, and include the terminator.

```matlab
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.

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

See Also

Functions

`fgets`, `fopen`, `instrhelp`, `query`
Properties

BytesAvailable, EOSCharCode, EOSMode, InputBufferSize, Status, Terminator, Timeout, ValuesReceived
**Purpose**
Remove data from input buffer

**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.

**Remarks**
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.

**See Also**

**Functions**
clrdevice, flushoutput, fopen, readasync

**Properties**
BytesAvailable
**Purpose**
Remove data from output buffer

**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.

**Remarks**
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 M-file 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.

**See Also**

**Functions**
clrdevice, flushinput, fopen, fprintf, fwrite

**Properties**
BytesToOutput, OutputEmptyFcn
Purpose 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.

Remarks 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, 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.
**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.

```matlab
    g = gpib('ni',0,1);
fopen(g)
fprintf(g,'*IDN?')
idn = fscanf(g);
fclose(g)
```

**See Also**

**Functions**

`fclose`, `instrhelp`, `propinfo`

**Properties**

`BytesAvailable`, `BytesToOutput`, `Status`, `ValuesReceived`, `ValuesSent`
Purpose  Write text to instrument

Syntax

\begin{verbatim}
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
\end{verbatim}

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>An interface object.</td>
</tr>
<tr>
<td>'cmd'</td>
<td>The string written to the instrument.</td>
</tr>
<tr>
<td>'format'</td>
<td>C language conversion specification.</td>
</tr>
<tr>
<td>'mode'</td>
<td>Specifies whether data is written synchronously or asynchronously.</td>
</tr>
</tbody>
</table>

Description

\texttt{fprintf(obj,'cmd')} writes the string \texttt{cmd} to the instrument connected to \texttt{obj}. The default format is \texttt{%s\n}. The write operation is synchronous and blocks the command line until execution is complete.

\texttt{fprintf(obj,'format','cmd')} writes the string using the format specified by \texttt{format}.

\texttt{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 \texttt{sprintf} file I/O format specifications or a C manual for more information.

\texttt{fprintf(obj,'cmd','mode')} writes the string with command-line access specified by \texttt{mode}. If \texttt{mode} is \texttt{sync}, \texttt{cmd} is written synchronously and the command line is blocked. If \texttt{mode} is \texttt{async}, \texttt{cmd} is written asynchronously and the command line is not blocked. If \texttt{mode} is not specified, the write operation is synchronous.

\texttt{fprintf(obj,'format','cmd','mode')} writes the string using the specified format. If \texttt{mode} is \texttt{sync}, \texttt{cmd} is written synchronously. If \texttt{mode} is \texttt{async}, \texttt{cmd} is written asynchronously.
Remarks

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 and a hardware connection is not detected.

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 M-file 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.

Synchronous and asynchronous write operations are discussed in more detail in “Synchronous Versus Asynchronous Write Operations” on page 3-17.

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.

**Examples**

Create the serial port object s, connect s 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.

```plaintext
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?')

See Also

Functions
fopen, fwrite, instrhelp, query, sprintf

Properties
BytesToOutput, EOSCharCode, EOSMode, OutputBufferSize, OutputEmptyFcn, Status, TransferStatus, ValuesSent
Purpose
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.

\[ A = \text{fread}(\text{obj}, \text{size}, 'precision') \] 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 Remarks.

\[ [A, \text{count}] = \text{fread}(...) \] returns the number of values read to count.

\[ [A, \text{count}, \text{msg}] = \text{fread}(...) \] returns a warning message to msg if the read operation was unsuccessful.

\[ [A, \text{count}, \text{msg}, \text{datagramaddress}] = \text{fread}(\text{obj}, \ldots) \] returns the datagram address to datagramaddress if obj is a UDP object. If more than one datagram is read, datagramaddress is ‘’.

\[ [A, \text{count}, \text{msg}, \text{datagramaddress}, \text{datagramport}] = \text{fread}(\text{obj}, \ldots) \] returns the datagram port to datagramport if obj is a UDP object. If more than one datagram is read, datagramport is [ ].

**Remarks**

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.

**Rules for Completing a Binary Read Operation**

A read operation with `fread` blocks access to the MATLAB command line 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

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Precision</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>uchar</td>
<td>8-bit unsigned character</td>
</tr>
<tr>
<td></td>
<td>schar</td>
<td>8-bit signed character</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>8-bit signed or unsigned character</td>
</tr>
<tr>
<td>Integer</td>
<td>int8</td>
<td>8-bit integer</td>
</tr>
<tr>
<td></td>
<td>int16</td>
<td>16-bit integer</td>
</tr>
<tr>
<td></td>
<td>int32</td>
<td>32-bit integer</td>
</tr>
<tr>
<td></td>
<td>uint8</td>
<td>8-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>uint16</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>uint32</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>short</td>
<td>16-bit integer</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>32-bit integer</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>32- or 64-bit integer</td>
</tr>
<tr>
<td></td>
<td>ushort</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>uint</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>ulong</td>
<td>32- or 64-bit unsigned integer</td>
</tr>
<tr>
<td>Floating-point</td>
<td>single</td>
<td>32-bit floating point</td>
</tr>
<tr>
<td></td>
<td>float32</td>
<td>32-bit floating point</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>32-bit floating point</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>64-bit floating point</td>
</tr>
<tr>
<td></td>
<td>float64</td>
<td>64-bit floating point</td>
</tr>
</tbody>
</table>

### See Also

- fgetl
- fgets
- fopen
- fscanf
- instrhelp

### Functions

- fgetl
- fgets
- fopen
- fscanf
- instrhelp
Properties
BytesAvailable, InputBufferSize, Status, ValuesReceived
Purpose
Read data from instrument, and format as text

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 [ ].

Remarks

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.

**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.
Examples

Create the serial port object `s` and connect `s` to a Tektronix TDS 210 oscilloscope, which is displaying a sine wave.

```matlab
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.

```matlab
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.

```matlab
s.BytesAvailable
ans =
    13
```

Use `fscanf` to read the measurement type. The operation will complete when the first terminator is read.

```matlab
meas = fscanf(s)
meas =
    PK2PK
```

Use `fscanf` to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```matlab
pk2pk = fscanf(s,'%e',6)
pk2pk =
    2.0200
```
Disconnect s from the scope, and remove s from memory and the workspace.

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

### See Also

**Functions**

`fgetl`, `fgets`, `fopen`, `fread`, `instrhelp`, `scanstr`, `sscanf`

**Properties**

`BytesAvailable`, `BytesAvailableFcn`, `EOSCharCode`, `EOSMode`, `InputBufferSize`, `Status`, `Terminator`, `Timeout`, `TransferStatus`
Purpose
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 18-45.

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.
Remarks

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 and a hardware connection is not detected.

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 M-file 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.

Synchronous and asynchronous write operations are discussed in more detail in “Synchronous Versus Asynchronous Write Operations” on page 3-17.

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.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Precision</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>uchar</td>
<td>8-bit unsigned character</td>
</tr>
<tr>
<td></td>
<td>schar</td>
<td>8-bit signed character</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>8-bit signed or unsigned character</td>
</tr>
<tr>
<td>Integer</td>
<td>int8</td>
<td>8-bit integer</td>
</tr>
<tr>
<td></td>
<td>int16</td>
<td>16-bit integer</td>
</tr>
<tr>
<td></td>
<td>int32</td>
<td>32-bit integer</td>
</tr>
<tr>
<td></td>
<td>uint8</td>
<td>8-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>uint16</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>uint32</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>short</td>
<td>16-bit integer</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td>32-bit integer</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>32- or 64-bit integer</td>
</tr>
<tr>
<td></td>
<td>ushort</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>uint</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td></td>
<td>ulong</td>
<td>32- or 64-bit unsigned integer</td>
</tr>
</tbody>
</table>
See Also

Functions

fopen, fprintf, instrhelp

Properties

OutputBufferSize, OutputEmptyFcn, Status, Timeout, TransferStatus, ValuesSent
Purpose
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 strings 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.

Remarks
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');
get

```matlab
out = get(g,'eosmode');
out = get(g,'EOSM');
```

**Examples**
This example illustrates some of the ways you can use `get` to return property values for the GPIB object `g`.

```matlab
g = gpib('ni',0,1);
out1 = get(g);
out2 = get(g,{'PrimaryAddress','EOSCharCode'});
get(g,'EOIMode')
anse =
on
```

**See Also**

**Functions**

`instrhelp`, `propinfo`, `set`
**Purpose**
Check and return error message from instrument

**Syntax**
\[ \text{msg} = \text{geterror}(\text{obj}) \]

**Arguments**
- **obj**
  A device object.
- **msg**
The error message returned from the instrument.

**Description**
\[ \text{msg} = \text{geterror}(\text{obj}) \]
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.
**Purpose**
Create GPIB object

**Syntax**

```matlab
obj = gpib('vendor', boardindex, primaryaddress)
```

```matlab
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.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>advantech</td>
<td>Advantech Company hardware</td>
</tr>
<tr>
<td>agilent</td>
<td>Agilent Technologies hardware</td>
</tr>
<tr>
<td>cec</td>
<td>Capital Equipment Corporation hardware</td>
</tr>
<tr>
<td>contec</td>
<td>CONTEC hardware</td>
</tr>
<tr>
<td>ics</td>
<td>ICS Electronics hardware</td>
</tr>
<tr>
<td>iotech</td>
<td>IOTech hardware</td>
</tr>
<tr>
<td>keithley</td>
<td>Keithley Instruments hardware</td>
</tr>
</tbody>
</table>
Vendor | Description
--- | ---
mcc | Measurement Computing Corporation hardware
ni | National Instruments hardware

```matlab
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.
```

**Remarks**

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:

```matlab
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.

### 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.

```matlab
g1 = gpib('ni',0,1);
fopen(g1)
```

The `Type`, `Name`, `BoardIndex`, and `PrimaryAddress` properties are automatically configured.

```matlab
get(g1, {'Type','Name','BoardIndex','PrimaryAddress'})
ans =
    'gpib'    'GPIB0-1'     [0]     [1]
```

To specify the secondary address during object creation,

```matlab
g2 = gpib('ni',0,1,'SecondaryAddress',96);
```

### See Also

**Functions**

`fopen, instrhelp, instrhwinfo`
Properties

BoardIndex, Name, PrimaryAddress, SecondaryAddress, Status, Type
icdevice

**Purpose**
Create device object

**Syntax**

```matlab
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 VXIplug&play and Interchangeable Virtual Instrument (IVI) drivers. To use these drivers, you must first have a MATLAB instrument driver wrapper for the underlying VXIplug&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 VXIplug&play instrument driver or MATLAB IVI instrument driver. Communication to the instrument is done through the resource specified by rsrcname. For example, all VXIplug&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 string 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:

```matlab
d = icdevice('tektronix_tds210',g,'ObjectVisibility','on');
d = icdevice('tektronix_tds210',g,'objectvisibility','on');
d = icdevice('tektronix_tds210',g,'ObjectVis','on');
```

**Remarks**

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:
icdevice

- **Interface** specifies the interface used to communicate with the instrument. For device objects created using interface objects, it is that interface object. For VXIplug&play and IVI-C, this is the session handle to the driver session. For IVI-COM and 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 VXIplug&play and IVI drivers. For MATLAB interface drivers, RsrcName is an empty string.

- **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.

### Examples

The first example creates a device object for a Tektronix TDS 210 oscilloscope that is connected to a Keithley GPIB board, using a MATLAB interface object and MATLAB interface instrument driver.

```matlab
g = gpib('keithley',0,2);
d = icdevice('tektronix_tds210',g);
```

Connect to the instrument.

```matlab
connect(d);
```

List the oscilloscope settings that can be configured.

```matlab
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);
icdevice

See Also

Functions
icdevice/connect, icdevice/disconnect, instrhelp

Properties
Status
Purpose: 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.

Remarks: 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.
# Property Inspector: Device

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstrumentModel</td>
<td>[1x1 GFIB]</td>
</tr>
<tr>
<td>Language</td>
<td>english</td>
</tr>
<tr>
<td>LogicalName</td>
<td>GPB1-4</td>
</tr>
<tr>
<td>Math</td>
<td>ch1, ch2</td>
</tr>
<tr>
<td>Measurement</td>
<td>[1x1 Measurement]</td>
</tr>
<tr>
<td>name</td>
<td>scope-tektronix_L210</td>
</tr>
<tr>
<td>DisplayVisibility</td>
<td>on</td>
</tr>
<tr>
<td>ResponseHeader</td>
<td>off</td>
</tr>
<tr>
<td>RunName</td>
<td></td>
</tr>
<tr>
<td>Status</td>
<td>closed</td>
</tr>
<tr>
<td>Tag</td>
<td></td>
</tr>
<tr>
<td>Timeout</td>
<td>100</td>
</tr>
<tr>
<td>Trigger</td>
<td>[1x1 Trigger]</td>
</tr>
<tr>
<td>Type</td>
<td>scope</td>
</tr>
<tr>
<td>UserData</td>
<td>[0x0 double swap]</td>
</tr>
<tr>
<td>VerboseEnabled</td>
<td>off</td>
</tr>
<tr>
<td>Waveform</td>
<td>[1x1 Waveform]</td>
</tr>
</tbody>
</table>

18-70
**Purpose**
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.

**Remarks**
You should use `instrcallback` as a template from which you create callback functions that suit your specific application needs.

**Examples**
The following example creates the serial port objects `s`, and configures `s` to execute `instrcallback` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

```matlab
s = serial('COM1');
set(s,'OutputEmptyFcn',@instrcallback)
fclose(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.

```c
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```
**Purpose**
Read instrument objects from memory to MATLAB workspace

**Syntax**
```matlab
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`.

**Remarks**
`instrfind` will not return an instrument object if its `ObjectVisibility` property is configured to `off`.

You must specify property values using the same format as the `get` function returns. For example, if `get` returns the `Name` property value 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 string 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 string values.

You can use property name/property value string pairs, structures, and cell array pairs in the same call to instrfind.

**Examples**

Suppose you create the following two GPIB objects.

```matlab
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.

```matlab
out1 = instrfind('Type','gpib');
out2 = instrfind({'Type','EOSCharCode'},{'gpib','CR'});
```

You can also use instrfind to return cleared instrument objects to the MATLAB workspace.

```matlab
clear g1 g2
newobjs = instrfind
```

**Instrument Object Array**

<table>
<thead>
<tr>
<th>Index</th>
<th>Type</th>
<th>Status</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>gpib</td>
<td>open</td>
<td>GPIBO-1</td>
</tr>
<tr>
<td>2</td>
<td>gpib</td>
<td>open</td>
<td>GPIBO-2</td>
</tr>
</tbody>
</table>

Assign the instrument objects their original names.

```matlab
g1 = newobjs(1);
g2 = newobjs(2);
```
Close both g1 and g2.

fclose(newobjs)

See Also

Functions

clear, get, instrfindall, propinfo

Properties

ObjectVisibility
### instrfindall

#### Purpose
Find visible and hidden instrument objects

#### Syntax
```matlab
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`. 

18-76
Note that you can use string property name/property value pairs, structures, and cell array property name/property value pairs in the same call to instrfindall.

Remarks

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 returned by the get function. For example, if get returns the Name property value 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 string 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 string values.

Examples

Suppose you create the following instrument objects.

```matlab
s1 = serial('COM1');
s2 = serial('COM2');
g1 = gpib('keithley',0,2);
set(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
```

<table>
<thead>
<tr>
<th>Instrument Object Array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Index</strong>: 1</td>
</tr>
<tr>
<td><strong>Index</strong>: 2</td>
</tr>
<tr>
<td><strong>Index</strong>: 3</td>
</tr>
</tbody>
</table>

The following statements use `instrfindall` to return objects with specific property settings, which are passed as cell arrays:

```matlab
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:

```matlab
set(instrfindall,'ObjectVisibility','on')
```

See Also

- **Functions**
  - `instrfind`, `propinfo`

Properties

- `ObjectVisibility`
Purpose
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
'sequence' 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.
**Remarks**

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.

**Examples**

The following commands illustrate some of the ways you can get function and property help without creating an instrument object.

```
instrhelp gpib
go = instrhelp('gpib.m');
instrhelp set
instrhelp('gpib/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');
```

**See Also**

**Functions**

propinfo
Purpose
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 or vxipnp.
'adaptor'     A supported GPIB or VISA adaptor.
'drivername'  Name of VXIplug&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, MATLAB version, and supported interfaces.

out = instrhwinfo('interface') returns information related to the interface specified by interface. interface can be serial, gpib,
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 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.

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 advantech, agilent, cec, contec, ics, iotech,
keithley, mcc, and ni. The VISA adaptors are agilent, ni, 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 agilent, ni, 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 string or a cell array of strings. 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.

Remarks
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.

Examples
The following commands illustrate some of the ways you can get hardware-related information without creating an instrument object.

```matlab
out1 = instrhwinfo;
out2 = instrhwinfo('serial');
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.

```matlab
vs = visa('agilent','ASRL1::INSTR');
out5 = instrhwinfo(vs)
```

```matlab
out5 =
    AdaptorDllName: [1x67 char]
    AdaptorDllVersion: 'Version 1.2 (R13)
    AdaptorName: 'AGILENT'
    VendorDriverDescription: 'Agilent Technologies VISA Driver'
    VendorDriverVersion: '1.1000'
```

```matlab
vsdll = instrhwinfo(vs,'AdaptorDllName')
vsdll = D:\V6\toolbox\instrument\instrumentadaptors\win32\mwagilentvisa.dll
```
**Purpose**
Define and retrieve commands that identify instruments

**Syntax**

```matlab
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`.

**Remarks**

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 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.

**Examples**

Set the identification command to `*ID?`.

```matlab
instrid('*ID?')```
Specify three new identification commands using a cell array.

\[
\text{instrid}({'*IDN?', '*ID?', 'IDEN?'})
\]

Assign a list of current identification commands to an output variable.

\[
id\_commands = \text{instrid};
\]

See Also

\begin{itemize}
\item instrhwinfo
\item tmtool
\end{itemize}
**Purpose**
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, string, 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 string 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

```matlab
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`.

**Remarks**

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.

Examples

    instrnotify('PropertyChangedPostSet', @instrcallback);
    g = gpib('keithley', 0, 5);
    set(g, 'Name', 'mygpb');
    fopen(g);
    fclose(g);
    instrnotify('PropertyChangedPostSet',@instrcallback,'-remove');
Purpose
Disconnect and delete all instrument objects

Syntax
instrreset

Description
instrreset disconnects and deletes all instrument objects.

Remarks
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
Functions

clear, delete, fclose, isvalid, stopasync
**Purpose**
Execute driver-specific function on device object

**Syntax**
```matlab
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`.

**Remarks**
To list the driver-specific functions supported by `obj`, type
```matlab
methods(obj)
```
To display help for a specific function, type
```matlab
instrhelp(obj,'name')
```

**Examples**
Create a device object for a Tektronix TDS 210 oscilloscope that is connected to a National Instruments GPIB board.
```matlab
g = gpib('ni',0,2);
d = icdevice('tektronix_tds210',g);
```
Perform a self-calibration for the oscilloscope by invoking the `calibrate` function.
```matlab
out = invoke(d,'calibrate')
```
out = '0'

0 indicates that the self-calibration completed without any errors.

See Also

See Also Functions
instrhelp, methods

Properties
Status
isvalid

**Purpose**
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.

**Remarks**
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.

**Examples**
Suppose you create the following two GPIB objects:

```matlab
g1 = gpib('ni',0,1);
g2 = gpib('ni',0,2);
```

g2 becomes invalid after it is deleted.

```matlab
delete(g2)
```

isvalid verifies that g1 is valid and g2 is invalid.

```matlab
garray = [g1 g2];
isvalid(garray)
ans =
    1     0
```

**See Also**
- Functions:
  - clear, delete

18-92
iviconfigurationstore

**Purpose**
Create IVI configuration store object

**Syntax**

```plaintext
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**

- Functions
  - `iviconfigurationstore/add`, `iviconfigurationstore/commit`,
  - `iviconfigurationstore/remove`, `iviconfigurationstore/update`
# length

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Length of instrument object array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><code>length(obj)</code></td>
</tr>
<tr>
<td>Arguments</td>
<td>obj An instrument object or an array of instrument objects.</td>
</tr>
<tr>
<td>Description</td>
<td><code>length(obj)</code> returns the length of obj. It is equivalent to the command <code>max(size(obj))</code>.</td>
</tr>
<tr>
<td>See Also</td>
<td><strong>Functions</strong> instrhelp, size</td>
</tr>
</tbody>
</table>
Purpose
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.

Remarks
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.

Examples
Suppose you create the GPIB objects g1 and g2, configure a few properties for g1, and connect both objects to their associated instruments.

```matlab
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.

get([g1 g2],'Status')
ans =
    'open'
    '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'

See Also

Functions

instrhelp, propinfo, save
makemid

**Purpose**
Convert driver to MATLAB instrument driver format

**Syntax**
- `makemid('driver')`
- `makemid('driver', 'filename')`
- `makemid('driver', 'type')`
- `makemid('driver', 'filename', 'type')`
- `makemid('driver', 'type', 'interface')`
- `makemid('driver', 'filename', 'type', 'interface')`

**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.
- `'interface'` The IVI-COM interface to be used.

**Description**
makemid('driver') searches through known driver types for `driver` and creates a MATLAB instrument driver representation of the driver. Known driver types include VXIplug&play, IVI-C, and IVI-COM. For `driver` you can use a Module (for IVI-C), a ProgramID (for IVI-COM), a LogicalName (for either IVI-C or IVI-COM), or the original VXIplug&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, ivi-c, and ivi-com.
makemid('driver', 'type', 'interface') and makemid('driver', 'filename', 'type', 'interface') specify the IVI-COM driver interface to be used for the object. type must be ivi-com.

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.

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

Functions

icdevice, midedit
**Purpose**  
Map memory for low-level memory read and write operations

**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).

**Remarks**  
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.

**Examples**  
Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

```plaintext
vv = visa('agilent','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)

See Also

Functions

fopen, fclose, mempeek, mempoke, memunmap

Properties

MappedMemorySize, Status
Purpose  
Low-level memory read from VXI register

Syntax

\[ out = \text{mempeek}(\text{obj}, \text{offset}) \]
\[ out = \text{mempeek}(\text{obj}, \text{offset}, 'precision') \]

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.

Remarks

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.

### Examples

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

```matlab
vv = visa('agilent','VXI0::130::INSTR');
fopen(vv)
```

Use `memmap` to map 16 bytes in the A16 address space.

```matlab
memmap(vv,'A16',0,16)
```

Perform a low-level read of the first and second instrument registers.

```matlab
reg1 = mempeek(vv,0,'uint16')
reg1 = 53247
reg2 = mempeek(vv,2,'uint16')
reg2 = 20993
```

Unmap the memory and disconnect `vv` from the instrument.

```matlab
memunmap(vv)
fclose(vv)
```

Refer to “Example: Using High-Level Memory Functions” on page 5-16 for a description of the first four registers of the E1432A digitizer.

### See Also

**Functions**

- `fopen`
- `memmap`
- `mempoke`
- `memunmap`

**Properties**

- `MappedMemorySize`
- `MemoryIncrement`
- `Status`
Purpose

Low-level memory write to VXI register

Syntax

mempoke(obj, data, offset)
mempoke(obj, data, offset, 'precision')

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A VISA-VXI or VISA-GPIB-VXI object.</td>
</tr>
<tr>
<td>data</td>
<td>The data written to the memory address.</td>
</tr>
<tr>
<td>offset</td>
<td>The offset in the mapped memory space to which the data is written.</td>
</tr>
<tr>
<td>'precision'</td>
<td>The number of bits to write to the memory address.</td>
</tr>
</tbody>
</table>

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.

Remarks

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.

**Examples**

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

```matlab
vv = visa('agilent','VXI0::130::INSTR');
fopen(vv)
```

Use `memmap` to map 16 bytes in the A16 address space.

```matlab
memmap(vv,'A16',0,16)
```

Perform a low-level write to the fourth instrument register, which has an offset of 6.

```matlab
mempoke(vv,45056,6,'uint16')
```

Unmap the memory and disconnect `vv` from the instrument.

```matlab
memunmap(vv)
fclose(vv)
```

Refer to “Example: Using High-Level Memory Functions” on page 5-16 for a description of the first four registers of the E1432A digitizer.

**See Also**

**Functions**

`fopen`, `memmap`, `mempeek`

**Properties**

`MappedMemorySize`, `MemoryIncrement`, `Status`
Purpose
High-level memory read from VXI register

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.
memread

out = memread(obj,offset,'precision','adrspace',size) reads a block of data with a size specified by size.

Remarks
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.

Examples
Create the VISA-VXI object vv associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

vv = visa('agilent','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)

Refer to “Example: Using High-Level Memory Functions” on page 5-16 for a description of the first four registers of the E1432A digitizer.

See Also

Functions
fopen, mempeek, memwrite
Properties

MemoryIncrement, MemorySpace, Status
memunmap

**Purpose**
Unmap memory for low-level memory read and write operations

**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.

**Remarks**
When the memory space is unmapped, the MappedMemorySize property is set to 0 and the MappedMemoryBase property is set to 0H.

**Examples**
Create the VISA-VXI object vv associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

```matlab
vv = visa('agilent','VXI0::130::INSTR');
fopen(vv)
memmap(vv,'A16',0,16)
```

Map 16 bytes in the A16 address space.

```matlab
memmap(vv,'A16',0,16)
```

Read the first and second instrument registers.

```matlab
reg1 = mempeek(vv,0,'uint16');
reg2 = mempeek(vv,2,'uint16');
```

Use memunmap to unmap the memory, and disconnect vv from the instrument.

```matlab
memunmap(vv)
fclose(vv)
```

**See Also**
**Functions**
memmap, mempeek, mempoke
Properties

MappedMemoryBase, MappedMemorySize
memwrite

Purpose
High-level memory write to VXI register

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.

Remarks

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.

**Examples**

Create the VISA-VXI object `vv` associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

```matlab
vv = visa('agilent','VXI0::130::INSTR');
fopen(vv)
```

Perform a high-level write to the fourth instrument register, which has an offset of 6.

```matlab
memwrite(vv,45056,6,'uint16','A16')
```

Disconnect `vv` from the instrument.

```matlab
fclose(vv)
```

Refer to “Example: Using High-Level Memory Functions” on page 5-16 for a description of the first four registers of the E1432A digitizer.

**See Also**

**Functions**

`fopen`, `memread`, `mempoke`

**Properties**

`MemoryIncrement`, `MemorySpace`, `Status`
**Purpose**
Class method names and descriptions

**Syntax**

```matlab
m = methods('classname')
m = methods(object)
m = methods(...,'-full')
```

**Arguments**

- `m` Cell array of strings
- `'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 strings, 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.

**Remarks**

`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**

**Functions**

`methodsview, what, which, help`
Purpose
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 VXIplug&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 VXIplug&play or IVI driver. This allows you to customize the behavior of device objects that use the VXIplug&play or IVI driver.

For details and examples on the MATLAB Instrument Driver Editor, see Chapter 14, “Using the Instrument Driver Editor”.

See Also
icdevice, makemid, midtest, tmtool
**midtest**

**Purpose**
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 array 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 Chapter 15, “Using the Instrument Driver Testing Tool”.

**Examples**
midtest('test.xml')

opens the MATLAB Instrument Driver Testing Tool with the test test.xml loaded.

**See Also**
icdevice, makemid, midedit, tmtool
**Purpose**

Convert instrument object to MATLAB code

**Syntax**

```matlab
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')`

`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.

**Remarks**

You can recreate a saved instrument object by typing the name of the M-file at the MATLAB command line.

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 M-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`.

**Examples**

Suppose you create the GPIB object `g`, and configure several property values.

```matlab
   g = gpib('ni',0,1);
   set(g,'Tag','MyGPIB object','EOSMode','read','EOSCharCode','CR')
   set(g,'UserData',{'test',2,magic(10)})
```

18-116
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 =
    'test'        [2]      [10x10 double]

See Also

Functions

propinfo
**PropInfo**

**Purpose**
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.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td>The property data type. Possible values are any, ASCII value, callback, instrument range value, double, string, and struct.</td>
</tr>
<tr>
<td><strong>Constraint</strong></td>
<td>The type of constraint on the property value. Possible values are ASCII value, bounded, callback, instrument range value, enum, and none.</td>
</tr>
<tr>
<td><strong>ConstraintValue</strong></td>
<td>Property value constraint. The constraint can be a range of valid values or a list of valid string values.</td>
</tr>
<tr>
<td><strong>DefaultValue</strong></td>
<td>The property default value.</td>
</tr>
<tr>
<td><strong>ReadOnly</strong></td>
<td>The condition under which a property is read-only. Possible values are always, never, whileOpen, and whileRecording.</td>
</tr>
<tr>
<td><strong>Interface Specific</strong></td>
<td>If the property is interface-specific, a 1 is returned. If a 0 is returned, the property is supported for all interfaces.</td>
</tr>
</tbody>
</table>
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.

**Remarks**

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.

```matlab
out = propinfo(g,'EOSMode');
out = propinfo(g,'eosmode');
out = propinfo(g,'EOSM');
```

**Examples**

To return all property information for the GPIB object g,

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

To display all the property information for the InputBufferSize property,

```matlab
out.InputBufferSize
ans =
    Type: 'double'
    Constraint: 'none'
    ConstraintValue: ''
    DefaultValue: 512
    ReadOnly: 'whileOpen'
    InterfaceSpecific: 0
```
To display the default value for the EOSMode property,

```matlab
out.EOSMode.DefaultValue
ans =
none
```

**See Also**

**Functions**

get, instrhelp, set

18-120
Purpose
Write text to instrument, and read data from instrument

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.

Remarks

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.

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.

```matlab
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)
```
See Also

Functions
fopen, fprintf, fgets, sprintf

Properties
Status
Purpose
Read data asynchronously from instrument

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.

Remarks
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 an M-file callback function when the terminator or the specified amount of data is read.

**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.

**Examples**

This example creates the serial port object `s`, connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` 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.

```matlab
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
```
Initially, there is no data in the input buffer.

```matlab
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`.

```matlab
readasync(s)
s.BytesAvailable
ans =
   15
go = fscanf(s)
go =
   2.0399999619E0
fclose(s)
```

### See Also

**Functions**

- `fopen`
- `stopasync`

**Properties**

- `BytesAvailable`
- `BytesAvailableFcn`
- `ReadAsyncMode`
- `Status`
- `TransferStatus`
Purpose
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.

Remarks
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 12-6.

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';
record

```matlab
  g.RecordName = 'MyGPIBFile.txt';

  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)

  See Also

  Functions
  fclose, fopen, propinfo

  Properties
  RecordMode, RecordName, RecordStatus, Status
```
Purpose
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
Functions
iviconfigurationstore, iviconfigurationstore/add,
iviconfigurationstore/commit, iviconfigurationstore/update
**Purpose**  
Network name or network address

**Syntax**

```matlab
name = resolvehost('host')
[name,address] = resolvehost('host')
out = resolvehost('host','returntype')
```

**Arguments**

- `'host'`  
The network name or network address of host.
- `'returntype'`  
Return either the name or address of host
- `name`  
Network name of host
- `address`  
Network address of host

**Description**

`name = resolvehost('host')` returns the name of the specified host. You can specify `host` as either a network name or a network address. For example, `www.mathworks.com` is a network name and `144.212.100.10` is a network address.

`[name,address] = resolvehost('host')` returns the name and address of the specified host.

`out = resolvehost('host','returntype')` returns the host name if `returntype` is `name` and returns the host address if `returntype` is `address`.

**Examples**

The following commands show how you can return the host name and address.

```matlab
[name,address] = resolvehost('144.212.100.10')
name = resolvehost('144.212.100.10','name')
address = resolvehost('www.mathworks.com','address')
```

**See Also**

**Functions**

`tcpip`, `udp`
**Purpose**
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`.

**Remarks**
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 strings. For example, to 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.
**Examples**

This example illustrates how to use the command form and the functional form of `save`.

```matlab
s = serial('COM1');
set(s,'BaudRate',2400,'StopBits',1)
save MySerial1 s
set(s,'BytesAvailableFcn',@mycallback)
save('MySerial2','s')
```

**See Also**

**Functions**

`instrhelp`, `load`, `propinfo`, `record`

**Properties**

`Status`
Purpose
Read data from instrument, format as text, and parse

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 string.

A = scanstr(obj,'delimiter') parses the data into separate variables based on the specified delimiter. delimiter can be a single character or a string array. If delimiter is a string 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 M-file 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 string) 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.
```

### Remarks

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.

### 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'
```
See Also

Functions
fopen, fscanf, instrhelp, sscanf, textread

Properties
EOSCharCode, EOSMode, Status, Terminator, ValuesReceived
Purpose
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.
Purpose
Create serial port object

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.

Remarks
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.

```matlab
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.

### Examples

This example creates the serial port object `s1` associated with the serial port COM1.

```matlab
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```matlab
get(s1,{'Type','Name','Port'})
ans =
    'serial'   'Serial-COM1'   'COM1'
```

To specify properties during object creation,

```matlab
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

See Also

- **Functions**
  - fclose, fopen, propinfo

- **Properties**
  - Name, Port, Status, Type
Purpose
Send break to instrument

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.

Remarks
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.

See Also
Functions
fopen, stopasync

Properties
Status
Purpose
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 string 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 string values.

props = set(obj,'PropertyName') returns the valid values for PropertyName to props. props is a cell array of possible string 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 strings 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.

Remarks
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')

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 ]
set

See Also

Functions

get, instrhelp, propinfo
Purpose
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.

See Also
Functions
instrhelp, length
**spoll**

**Purpose**
Perform serial poll

**Syntax**
```
out = spoll(obj)
out = 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 MATLAB.
- **out**  
The GPIB objects ready for servicing.

**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 line 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.

**Remarks**
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 service request (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.
Examples

If obj is a four-element array and val is set to [1 3], then spoll will block access to the MATLAB command line until the instruments connected to the first and third GPIB objects have both asserted their SRQ line, or a timeout occurs.

See Also

Functions

gpib, length

Properties

BoardIndex, Timeout
**stopasync**

**Purpose**
Stop asynchronous read and write operations

**Syntax**

```plaintext
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`.

**Remarks**

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.

**See Also**

**Functions**

- `fprintf`
- `fwrite`
- `readasync`
Properties

ReadAsyncMode, TransferStatus
**Purpose**
Create TCPIP object

**Syntax**

```
obj = tcpip('rhost')
obj = tcpip('rhost',rport)
obj = tcpip(...,'PropertyName',PropertyValue,...)
```

**Arguments**

- `'rhost'` The remote host.
- `rport` The remote port.
- `'PropertyName'` A TCPIP property name.
- `PropertyValue` A property value supported by `PropertyName`.
- `obj` The TCPIP object.

**Description**

`obj = tcpip('rhost')` creates a TCPIP object, `obj`, associated with remote host `rhost` and the default remote port value of 80.

`obj = tcpip('rhost',rport)` creates a TCPIP object with remote port value `rport`.

`obj = tcpip(...,'PropertyName',PropertyValue,...)` creates a TCPIP object with the specified property name/property value pairs. If an invalid property name or property value is specified, the object is not created.

**Remarks**
At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with TCPIP objects.

```
instrhelp tcpip
```

When you create a TCPIP object, these property values are automatically configured:

- *Type* is given by `tcpip`.
- *Name* is given by concatenating TCPIP with the remote host name specified in the `tcpip` function.
RemoteHost and RemotePort are given by the values specified in the tcpip 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.

\[
\begin{align*}
t &= \text{tcpip}('144.212.113.252', 'InputBufferSize', 1024); \\
t &= \text{tcpip}('144.212.113.252', 'inputbuffersize', 1024); \\
t &= \text{tcpip}('144.212.113.252', 'INPUT', 1024);
\end{align*}
\]

When the TCPIP object is constructed, the Status property value is closed. Once 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’s default. The LocalPort property defaults to a value of [ ] and it causes any free local port to be used. LocalPort is updated when fopen is issued.

**Examples**

Start a TCP/IP echo server and create a TCPIP object.

\[
\begin{align*}
echotcpip('on', 4012) \\
t &= \text{tcpip}('localhost', 4012);
\end{align*}
\]

Connect the TCPIP object to the host.

\[
fopen(t)
\]

Write to the host and read from the host.

\[
\begin{align*}
fwrite(t, 65:74) \\
A &= \text{fread}(t, 10);
\end{align*}
\]
Disconnect the TCPIP object from the host and stop the echo server.

```matlab
fclose(t)
echotcip('off')
```

**See Also**

**Functions**

fopen, sendmail, udp, urlread, urlwrite

**Properties**

LocalHost, LocalPort, LocalPortMode, Name, RemoteHost, RemotePort, Status, Type
Purpose

Open Test & Measurement Tool

Syntax

tmtool

Description

tmtool starts the Test & Measurement Tool. The Test & Measurement Tool displays the resources (hardware, drivers, interfaces, etc.) accessible to the toolboxes that support the tool, and enables you to configure and communicate with those resources.

You use the Test & Measurement Tool to manage your instrument control session. This tool enables you to

- Search for available hardware and drivers
- Create instrument objects
- Connect to an instrument
- Configure instrument settings
- Write data to an instrument
- Read data from an instrument
- Save a log of your session as an M-file

For a full description of the Test & Measurement Tool with examples, see Chapter 13, “Test & Measurement Tool”.

See Also

midedit, midtest
### Purpose
Send trigger message to instrument

### 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`.

### Remarks
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.

### See Also

#### Functions
- `fopen`

#### Properties
- `Status`, `TriggerLine`, `TriggerType`
Purpose
Create UDP object

Syntax

\[
\begin{align*}
\text{obj} & = \text{udp}('') \\
\text{obj} & = \text{udp}('rhost') \\
\text{obj} & = \text{udp}('rhost',\text{rport}) \\
\text{obj} & = \text{udp}(\ldots, '\text{PropertyName}', \text{PropertyValue}, \ldots)
\end{align*}
\]

Arguments

- 'rhost' The remote host.
- rport The remote port.
- 'PropertyName' A UDP property name.
- PropertyValue A property value supported by PropertyName.
- obj The UDP object.

Description

\[
\begin{align*}
\text{obj} = \text{udp}('') & \text{ creates a UDP object, obj, not associated with a remote host.} \\
\text{obj} = \text{udp}('rhost') & \text{ creates a UDP object associated with remote host rhost.} \\
\text{obj} = \text{udp}('rhost',\text{rport}) & \text{ creates a UDP object with remote port value, rport. The default remote port is 9090.} \\
\text{obj} = \text{udp}(\ldots, '\text{PropertyName}', \text{PropertyValue}, \ldots) & \text{ creates a UDP object with the specified property name/property value pairs. If an invalid property name or property value is specified, the object is not created.}
\end{align*}
\]

Remarks

At any time, you can use the instrhelp function to view a complete listing of properties and functions associated with UDP objects.

\[
\text{instrhelp udp}
\]

When you create a UDP object, these properties are automatically configured:

- Type is given by udp.
Name is given by concatenating UDP with the remote host name specified in the udp function.

RemoteHost and RemotePort are given by the values specified in the udp 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.

```
u = udp('144.212.113.252','InputBufferSize',1024);
u = udp('144.212.113.252','inputbuffersize',1024);
u = udp('144.212.113.252','INPUT',1024);
```

The UDP object must be bound to the local socket with the fopen function. The default remote port is 9090. The default local host in multihome hosts is the system’s default. The LocalPort property defaults to a value of [] and it causes any free local port to be used. LocalPort is updated when fopen is issued. When the UDP object is constructed, the 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 will be sent in packets of at most 4096 bytes.

**Examples**

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 read from the host.

```matlab
fwrite(u,65:74)
A = fread(u,10);
```

Stop the echo server and disconnect the UDP object from the host.

```matlab
echoudp('off')
fclose(u)
```

**See Also**

**Functions**

fopen

**Properties**

LocalHost, LocalPort, LocalPortMode, Name, RemoteHost, RemotePort, Status, Type
update

Purpose
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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
<td>A unique name for the driver session.</td>
</tr>
<tr>
<td>SoftwareModule</td>
<td>string</td>
<td>The name of a software module entry in the configuration store.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HardwareAsset</td>
<td>string</td>
<td>The name of a hardware asset entry in the configuration store.</td>
</tr>
<tr>
<td>Description</td>
<td>Any string</td>
<td>Description of driver session</td>
</tr>
<tr>
<td>VirtualNames</td>
<td>structure</td>
<td>A struct array containing virtual name mappings</td>
</tr>
<tr>
<td>Cache</td>
<td>on/off</td>
<td>Enable caching if the driver supports it.</td>
</tr>
<tr>
<td>DriverSetup</td>
<td>Any string</td>
<td>This value is software module dependent</td>
</tr>
<tr>
<td>InterchangeCheck</td>
<td>on/off</td>
<td>Enable driver interchangeability checking, if supported</td>
</tr>
<tr>
<td>QueryInstrStatus</td>
<td>on/off</td>
<td>Enable instrument status querying by the driver</td>
</tr>
<tr>
<td>RangeCheck</td>
<td>on/off</td>
<td>Enable extended range checking by the driver, if supported</td>
</tr>
<tr>
<td>RecordCoercions</td>
<td>on/off</td>
<td>Enable recording of coercions by the driver, if supported</td>
</tr>
<tr>
<td>Simulate</td>
<td>on/off</td>
<td>Enable simulation by the driver</td>
</tr>
</tbody>
</table>

Valid fields for HardwareAsset are

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
<td>A unique name for the hardware asset</td>
</tr>
<tr>
<td>Description</td>
<td>Any string</td>
<td>Description of hardware asset</td>
</tr>
<tr>
<td>IOResourceDescriptor</td>
<td>string</td>
<td>The I/O address of the hardware asset</td>
</tr>
</tbody>
</table>
update

Valid fields for LogicalName are

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>string</td>
<td>A unique name for the logical name</td>
</tr>
<tr>
<td>Description</td>
<td>Any string</td>
<td>Description of hardware asset</td>
</tr>
<tr>
<td>Session</td>
<td>string</td>
<td>The name of a driver session entry in the configuration store</td>
</tr>
</tbody>
</table>

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.

Examples

Update the Description parameter of the driver session named ScopeSession in the IVI configuration store object named c.

```plaintext
c = iviconfigurationstore;
update(c, 'DriverSession', 'ScopeSession', 'Description', ...
  'A session.');
```

See Also

Functions

- iviconfigurationstore
- iviconfigurationstore/add
- iviconfigurationstore/commit
- iviconfigurationstore/remove
Purpose
Create VISA object

Syntax
obj = visa('vendor','rsrchna')
obj =
visa('vendor','rsrchna','PropertyName',PropertyValue,
    ...)

Arguments
'vendor' A supported VISA vendor.
'rsrchna' 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','rsrchna') creates the VISA object obj with a resource name given by rsrchna for the vendor specified by vendor. If an invalid vendor or resource name is specified, an error is returned and the VISA object is not created. The supported values for vendor are given below.

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agilent</td>
<td>Agilent Technologies VISA</td>
</tr>
<tr>
<td>ni</td>
<td>National Instruments VISA</td>
</tr>
<tr>
<td>tek</td>
<td>Tektronix VISA</td>
</tr>
</tbody>
</table>

The format for rsrchna is given below for the supported VISA interfaces. The values indicated by brackets are optional. You can use the instrument’s VISA Alias for rsrchna.
The `rsrcname` parameters are described below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>board</td>
<td>Board index (optional — defaults to 0)</td>
</tr>
<tr>
<td>chassis</td>
<td>VXI chassis index (optional — defaults to 0)</td>
</tr>
<tr>
<td>interface_No</td>
<td>USB interface</td>
</tr>
<tr>
<td>lan_device_name</td>
<td>Local Area Network (LAN) device name (optional — defaults to inst0)</td>
</tr>
<tr>
<td>manid</td>
<td>Manufacturer ID of the USB instrument</td>
</tr>
<tr>
<td>model_code</td>
<td>Model code for the USB instrument</td>
</tr>
<tr>
<td>port_number</td>
<td>Serial port number (optional — defaults to 1)</td>
</tr>
<tr>
<td>primary_address</td>
<td>Primary address of the GPIB instrument</td>
</tr>
<tr>
<td>remote_host</td>
<td>Host name or IP address of the instrument</td>
</tr>
<tr>
<td>secondary_address</td>
<td>Secondary address of the GPIB instrument (optional — defaults to 0)</td>
</tr>
<tr>
<td>serial_No</td>
<td>Index of the instrument on the USB hub</td>
</tr>
<tr>
<td>VXI_logical_address</td>
<td>Logical address of the VXI instrument</td>
</tr>
</tbody>
</table>
obj = 
visa('vendor', 'rsrccname', 'PropertyName', PropertyValue, ...)
creates the VISA 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 VISA object is not created.

Remarks

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.
Creating a VISA-TCPIP Object
When you create a VISA-TCPIP object, these properties are automatically configured:

- Type is given by visa-tcpi.
- 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.
Create a VISA-serial object connected to serial port COM1 using National Instruments VISA interface.

\[ vs = \text{visa('ni','ASRL1::INSTR')} \]

Create a VISA-GPIB object connected to board 0 with primary address 1 and secondary address 30 using Agilent Technologies VISA interface.

\[ vg = \text{visa('agilent','GPIB0::1::30::INSTR')} \]

Create a VISA-VXI object connected to a VXI instrument located at logical address 8 in the first VXI chassis.

\[ vv = \text{visa('agilent','VXI0::8::INSTR')} \]

Create a VISA-GPIB-VXI object connected to a GPIB-VXI instrument located at logical address 72 in the second VXI chassis.

\[ vgv = \text{visa('agilent','GPIB-VXI1::72::INSTR')} \]

Create a VISA-RSIB object connected to an instrument configured with IP address 192.168.1.33.

\[ vr = \text{visa('ni','RSIB::192.168.1.33::INSTR')} \]

Create a VISA-TCPIP object connected to an instrument configured with IP address 216.148.60.170.

\[ vt = \text{visa('tek','TCPIP::216.148.60.170::INSTR')} \]

Create a VISA-USB object connected to a USB instrument with manufacturer ID 0x1234, model code 125, and serial number A22-5.

\[ vu = \text{visa('agilent','USB::0x1234::125::A22-5::INSTR')} \]

See Also: Functions

fclose, fopen, instrhelp, instrhwinfo
Properties

BoardIndex, ChassisIndex, InterfaceIndex, LANName, LogicalAddress, ManufacturerID, ModelCode, Name, Port, PrimaryAddress, RsrpName, SecondaryAddress, SerialNumber, Status, Type
## Properties — By Category

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Object Base Properties (p. 19-2)</td>
<td>Apply to all interface objects</td>
</tr>
<tr>
<td>Interface-Specific Properties (p. 19-5)</td>
<td>Apply to instrument objects of particular interface type</td>
</tr>
<tr>
<td>Device Object Base Properties (p. 19-13)</td>
<td>Apply to all device objects</td>
</tr>
<tr>
<td>IVI Configuration Store Object Properties (p. 19-14)</td>
<td>Apply to IVI configuration store objects</td>
</tr>
</tbody>
</table>
## Interface Object Base Properties

### Writing Data (p. 19-2)
- **BytesToOutput**: Number of bytes currently in output buffer
- **OutputBufferSize**: Specify size of output buffer in bytes
- **Timeout**: Specify waiting time to complete read or write operation
- **TransferStatus**: Status of whether asynchronous read or write operation is in progress
- **ValuesSent**: Total number of values written to instrument

### Reading Data
- **BytesAvailable**: Number of bytes available in input buffer
- **InputBufferSize**: Specify size of input buffer in bytes
- **Timeout**: Specify waiting time to complete read or write operation

### Recording Data (p. 19-3)
- **Record file parameters, status**

### Callbacks (p. 19-3)
- **M-file callback functions to execute when specific events occur**

### General Purpose (p. 19-4)
- **Object description, byte order, status**
**Interface Object Base Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransferStatus</td>
<td>Status of whether asynchronous read or write operation is in progress</td>
</tr>
<tr>
<td>ValuesReceived</td>
<td>Total number of values read from instrument</td>
</tr>
</tbody>
</table>

**Recording Data**

- **RecordDetail**
  Specify amount of information saved to record file
- **RecordMode**
  Specify whether data and event information are saved to one or to multiple record files
- **RecordName**
  Specify name of record file
- **RecordStatus**
  Status of whether data and event information are saved to record file

**Callbacks**

- **BytesAvailableFcn**
  Specify M-file callback function to execute when specified number of bytes are available in input buffer, or terminator is read
- **BytesAvailableFcnCount**
  Specify number of bytes that must be available in input buffer to generate bytes-available event
- **BytesAvailableFcnMode**
  Specify whether bytes-available event is generated after specified number of bytes are available in input buffer, or after terminator is read
- **ErrorFcn**
  Specify M-file callback function to execute when error event occurs
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputEmptyFcnc</td>
<td>Specify M-file callback function to execute when output buffer is empty</td>
</tr>
<tr>
<td>TimerFcnc</td>
<td>Specify M-file callback function to execute when predefined period passes</td>
</tr>
<tr>
<td>TimerPeriod</td>
<td>Specify period between timer events</td>
</tr>
</tbody>
</table>

**General Purpose**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteOrder</td>
<td>Specify byte order of instrument</td>
</tr>
<tr>
<td>Name</td>
<td>Specify descriptive name for instrument object</td>
</tr>
<tr>
<td>ObjectVisibility</td>
<td>Control access to instrument object</td>
</tr>
<tr>
<td>Status</td>
<td>Status of whether object is connected to instrument</td>
</tr>
<tr>
<td>Tag</td>
<td>Specify label to associate with instrument object</td>
</tr>
<tr>
<td>Type</td>
<td>Instrument object type</td>
</tr>
<tr>
<td>UserData</td>
<td>Specify data to associate with instrument object</td>
</tr>
</tbody>
</table>
## Interface-Specific Properties

<table>
<thead>
<tr>
<th>Interface</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIB (p. 19-5)</td>
<td>Communication address, bus configuration, status</td>
</tr>
<tr>
<td>Serial Port (p. 19-6)</td>
<td>Communication settings, data format, signals and pins</td>
</tr>
<tr>
<td>TCPIP (p. 19-7)</td>
<td>Host and port information, terminator character</td>
</tr>
<tr>
<td>UDP (p. 19-7)</td>
<td>Host and port information, datagrams, terminator character</td>
</tr>
<tr>
<td>VISA-GPIB (p. 19-8)</td>
<td>Resource name, communication address, bus configuration, status</td>
</tr>
<tr>
<td>VISA-GPIB-VXI (p. 19-8)</td>
<td>Resource name, VXI memory, chassis information, communication address, bus configuration, status</td>
</tr>
<tr>
<td>VISA-RSIB (p. 19-9)</td>
<td>Resource name, communication address, communication settings</td>
</tr>
<tr>
<td>VISA-Serial (p. 19-10)</td>
<td>Resource name, communication settings, data format, signals and pins</td>
</tr>
<tr>
<td>VISA-TCP/IP (p. 19-10)</td>
<td>Resource name, communication address, communication settings</td>
</tr>
<tr>
<td>VISA-USB (p. 19-11)</td>
<td>Resource name, communication address, communication settings</td>
</tr>
<tr>
<td>VISA-VXI (p. 19-11)</td>
<td>Resource name, VXI memory, chassis information, status</td>
</tr>
</tbody>
</table>

### GPIB

- **BoardIndex**
  - Specify index number of interface board

- **BusManagementStatus**
  - State of GPIB bus management lines
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompareBits</td>
<td>Specify number of bits that must match EOS character to complete read operation, or to assert EOI line</td>
</tr>
<tr>
<td>EOIMode</td>
<td>Specify if EOI line is asserted at end of write operation</td>
</tr>
<tr>
<td>EOSCharCode</td>
<td>Specify EOS character</td>
</tr>
<tr>
<td>EOSMode</td>
<td>Specify when EOS character is written or read</td>
</tr>
<tr>
<td>HandshakeStatus</td>
<td>State of GPIB handshake lines</td>
</tr>
<tr>
<td>PrimaryAddress</td>
<td>Specify primary address of GPIB instrument</td>
</tr>
<tr>
<td>SecondaryAddress</td>
<td>Specify secondary address of GPIB instrument</td>
</tr>
</tbody>
</table>

**Serial Port**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaudRate</td>
<td>Specify bit transmit rate</td>
</tr>
<tr>
<td>BreakInterruptFcn</td>
<td>Specify M-file callback function to execute when break-interrupt event occurs</td>
</tr>
<tr>
<td>DataBits</td>
<td>Specify number of data bits to transmit</td>
</tr>
<tr>
<td>DataTerminalReady</td>
<td>Specify state of DTR pin</td>
</tr>
<tr>
<td>FlowControl</td>
<td>Specify data flow control method to use</td>
</tr>
<tr>
<td>Parity</td>
<td>Specify type of parity checking</td>
</tr>
<tr>
<td>PinStatus</td>
<td>State of CD, CTS, DSR, and RI pins</td>
</tr>
<tr>
<td>PinStatusFcn</td>
<td>Specify M-file callback function to execute when CD, CTS, DSR, or RI pin changes state</td>
</tr>
<tr>
<td>Port</td>
<td>Specify platform-specific serial port name</td>
</tr>
</tbody>
</table>
## Interface-Specific Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReadAsyncMode</td>
<td>Specify whether asynchronous read operation is continuous or manual</td>
</tr>
<tr>
<td>RequestToSend</td>
<td>Specify state of RTS pin</td>
</tr>
<tr>
<td>StopBits</td>
<td>Specify number of bits used to indicate end of byte</td>
</tr>
<tr>
<td>Terminator</td>
<td>Specify terminator character</td>
</tr>
</tbody>
</table>

### TCPIP

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LocalHost</td>
<td>Specify local host</td>
</tr>
<tr>
<td>LocalPort</td>
<td>Specify local host port for connection</td>
</tr>
<tr>
<td>LocalPortMode</td>
<td>Specify local host port selection mode</td>
</tr>
<tr>
<td>ReadAsyncMode</td>
<td>Specify whether asynchronous read operation is continuous or manual</td>
</tr>
<tr>
<td>RemoteHost</td>
<td>Specify remote host</td>
</tr>
<tr>
<td>RemotePort</td>
<td>Specify remote host port for connection</td>
</tr>
<tr>
<td>Terminator</td>
<td>Specify terminator character</td>
</tr>
<tr>
<td>TransferDelay</td>
<td>Specify use of TCP segment transfer algorithm</td>
</tr>
</tbody>
</table>

### UDP

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DatagramAddress</td>
<td>IP dotted decimal address of received datagram sender</td>
</tr>
<tr>
<td>DatagramPort</td>
<td>Port number of datagram sender</td>
</tr>
<tr>
<td>DatagramReceivedFcn</td>
<td>Specify M-file callback function to execute when datagram is received</td>
</tr>
<tr>
<td>DatagramTerminateMode</td>
<td>Configure terminate read mode when reading datagrams</td>
</tr>
<tr>
<td>LocalHost</td>
<td>Specify local host</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LocalPort</td>
<td>Specify local host port for connection</td>
</tr>
<tr>
<td>LocalPortMode</td>
<td>Specify local host port selection mode</td>
</tr>
<tr>
<td>ReadAsyncMode</td>
<td>Specify whether asynchronous read operation is continuous or manual</td>
</tr>
<tr>
<td>RemoteHost</td>
<td>Specify remote host</td>
</tr>
<tr>
<td>RemotePort</td>
<td>Specify remote host port for connection</td>
</tr>
<tr>
<td>Terminator</td>
<td>Specify terminator character</td>
</tr>
</tbody>
</table>

**VISA-GPIB**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoardIndex</td>
<td>Specify index number of interface board</td>
</tr>
<tr>
<td>EOIMode</td>
<td>Specify if EOI line is asserted at end of write operation</td>
</tr>
<tr>
<td>EOSCharCode</td>
<td>Specify EOS character</td>
</tr>
<tr>
<td>EOSMode</td>
<td>Specify when EOS character is written or read</td>
</tr>
<tr>
<td>PrimaryAddress</td>
<td>Specify primary address of GPIB instrument</td>
</tr>
<tr>
<td>RsrcName</td>
<td>Resource name for VISA instrument</td>
</tr>
<tr>
<td>SecondaryAddress</td>
<td>Specify secondary address of GPIB instrument</td>
</tr>
</tbody>
</table>

**VISA-GPIB-VXI**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoardIndex</td>
<td>Specify index number of interface board</td>
</tr>
<tr>
<td>ChassisIndex</td>
<td>Specify index number of VXI chassis</td>
</tr>
<tr>
<td>EOIMode</td>
<td>Specify if EOI line is asserted at end of write operation</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>EOSCharCode</td>
<td>Specify EOS character</td>
</tr>
<tr>
<td>EOSMode</td>
<td>Specify when EOS character is written or read</td>
</tr>
<tr>
<td>LogicalAddress</td>
<td>Specify logical address of VXI instrument</td>
</tr>
<tr>
<td>MappedMemoryBase</td>
<td>Base memory address of mapped memory</td>
</tr>
<tr>
<td>MappedMemorySize</td>
<td>Size of mapped memory for low-level read and write operations</td>
</tr>
<tr>
<td>MemoryBase</td>
<td>Base address of A24 or A32 space</td>
</tr>
<tr>
<td>MemoryIncrement</td>
<td>Specify whether VXI register offset increments after data is transferred</td>
</tr>
<tr>
<td>MemorySize</td>
<td>Size of memory requested in A24 or A32 address space</td>
</tr>
<tr>
<td>MemorySpace</td>
<td>Address space used by instrument</td>
</tr>
<tr>
<td>PrimaryAddress</td>
<td>Specify primary address of GPIB instrument</td>
</tr>
<tr>
<td>RsrcName</td>
<td>Resource name for VISA instrument</td>
</tr>
<tr>
<td>SecondaryAddress</td>
<td>Specify secondary address of GPIB instrument</td>
</tr>
<tr>
<td>Slot</td>
<td>Slot location of VXI instrument</td>
</tr>
</tbody>
</table>

**VISA-RSIB**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOIMode</td>
<td>Specify if EOI line is asserted at end of write operation</td>
</tr>
<tr>
<td>RemoteHost</td>
<td>Specify remote host</td>
</tr>
</tbody>
</table>
**VISA-Serial**

- **BaudRate**: Specify bit transmit rate
- **DataBits**: Specify number of data bits to transmit
- **DataTerminalReady**: Specify state of DTR pin
- **FlowControl**: Specify data flow control method to use
- **Parity**: Specify type of parity checking
- **PinStatus**: State of CD, CTS, DSR, and RI pins
- **Port**: Specify platform-specific serial port name
- **ReadAsyncMode**: Specify whether asynchronous read operation is continuous or manual
- **RequestToSend**: Specify state of RTS pin
- **RsrcName**: Resource name for VISA instrument
- **StopBits**: Specify number of bits used to indicate end of byte
- **Terminator**: Specify terminator character

**VISA-TCPIP**

- **BoardIndex**: Specify index number of interface board
- **EOIMode**: Specify if EOI line is asserted at end of write operation
- **EOSCharCode**: Specify EOS character
- **EOSMode**: Specify when EOS character is written or read
- **LANName**: Specify LAN device name
- **RemoteHost**: Specify remote host
### VISA-USB

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoardIndex</td>
<td>Specify index number of interface board</td>
</tr>
<tr>
<td>EOIMode</td>
<td>Specify if EOI line is asserted at end of write operation</td>
</tr>
<tr>
<td>EOSCharCode</td>
<td>Specify EOS character</td>
</tr>
<tr>
<td>EOSMode</td>
<td>Specify when EOS character is written or read</td>
</tr>
<tr>
<td>InterfaceIndex</td>
<td>Specify USB interface number</td>
</tr>
<tr>
<td>ManufacturerID</td>
<td>Specify manufacturer ID of USB instrument</td>
</tr>
<tr>
<td>ModelCode</td>
<td>Specify model code of USB instrument</td>
</tr>
<tr>
<td>SerialNumber</td>
<td>Specify index of USB instrument on USB hub</td>
</tr>
</tbody>
</table>

### VISA-VXI

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChassisIndex</td>
<td>Specify index number of VXI chassis</td>
</tr>
<tr>
<td>EOIMode</td>
<td>Specify if EOI line is asserted at end of write operation</td>
</tr>
<tr>
<td>EOSCharCode</td>
<td>Specify EOS character</td>
</tr>
<tr>
<td>EOSMode</td>
<td>Specify when EOS character is written or read</td>
</tr>
<tr>
<td>InterruptFcn</td>
<td>Specify M-file callback function to execute when interrupt event occurs</td>
</tr>
<tr>
<td>LogicalAddress</td>
<td>Specify logical address of VXI instrument</td>
</tr>
<tr>
<td>MappedMemoryBase</td>
<td>Base memory address of mapped memory</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MappedMemorySize</td>
<td>Size of mapped memory for low-level read and write operations</td>
</tr>
<tr>
<td>MemoryBase</td>
<td>Base address of A24 or A32 space</td>
</tr>
<tr>
<td>MemoryIncrement</td>
<td>Specify whether VXI register offset increments after data is transferred</td>
</tr>
<tr>
<td>MemorySize</td>
<td>Size of memory requested in A24 or A32 address space</td>
</tr>
<tr>
<td>MemorySpace</td>
<td>Address space used by instrument</td>
</tr>
<tr>
<td>RsrcName</td>
<td>Resource name for VISA instrument</td>
</tr>
<tr>
<td>Slot</td>
<td>Slot location of VXI instrument</td>
</tr>
<tr>
<td>TriggerFcn</td>
<td>Specify M-file callback function to execute when trigger event occurs</td>
</tr>
<tr>
<td>TriggerLine</td>
<td>Specify trigger line on VXI instrument</td>
</tr>
<tr>
<td>TriggerType</td>
<td>Specify trigger type</td>
</tr>
</tbody>
</table>
**Device Object Base Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfirmationFcn</td>
<td>Specify M-file callback function to execute when confirmation event occurs</td>
</tr>
<tr>
<td>DriverType</td>
<td>Specify type of driver used to communicate with instrument</td>
</tr>
<tr>
<td>InstrumentModel</td>
<td>Instrument model that object connects to</td>
</tr>
<tr>
<td>Interface</td>
<td>Interface object that communicates with instrument</td>
</tr>
<tr>
<td>LogicalName</td>
<td>Specify description of interface used to communicate with instrument</td>
</tr>
<tr>
<td>Name</td>
<td>Specify descriptive name for instrument object</td>
</tr>
<tr>
<td>ObjectVisibility</td>
<td>Control access to instrument object</td>
</tr>
<tr>
<td>Status</td>
<td>Status of whether object is connected to instrument</td>
</tr>
<tr>
<td>Tag</td>
<td>Specify label to associate with instrument object</td>
</tr>
<tr>
<td>Timeout</td>
<td>Specify waiting time to complete read or write operation</td>
</tr>
<tr>
<td>Type</td>
<td>Instrument object type</td>
</tr>
<tr>
<td>UserData</td>
<td>Specify data to associate with instrument object</td>
</tr>
</tbody>
</table>
## IVI Configuration Store Object Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActualLocation</td>
<td>Configuration store file used by IVI configuration store object</td>
</tr>
<tr>
<td>DriverSessions</td>
<td>Array of driver sessions contained in IVI configuration store</td>
</tr>
<tr>
<td>HardwareAssets</td>
<td>Array of hardware assets contained in IVI configuration store</td>
</tr>
<tr>
<td>LogicalNames</td>
<td>Array of logical names contained in IVI configuration store</td>
</tr>
<tr>
<td>MasterLocation</td>
<td>Full pathname of master configuration store file</td>
</tr>
<tr>
<td>Name (iviconfigurationstore)</td>
<td>Name of IVI configuration server</td>
</tr>
<tr>
<td>ProcessLocation</td>
<td>Configuration store file for process to use if master configuration store is not used</td>
</tr>
<tr>
<td>PublishedAPIs</td>
<td>Array of published APIs in IVI configuration store</td>
</tr>
<tr>
<td>Revision</td>
<td>IVI configuration store version</td>
</tr>
<tr>
<td>ServerDescription</td>
<td>IVI configuration store server description</td>
</tr>
<tr>
<td>Sessions</td>
<td>Array of driver sessions in IVI configuration store</td>
</tr>
<tr>
<td>SoftwareModules</td>
<td>Array of software modules in IVI configuration store</td>
</tr>
<tr>
<td>SpecificationVersion</td>
<td>IVI configuration server specification version that this server revision complies with</td>
</tr>
<tr>
<td>Vendor</td>
<td>IVI configuration server vendor</td>
</tr>
</tbody>
</table>
Properties — Alphabetical List
**ActualLocation**

**Purpose**
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**: String

**Values**
The default value is the master configuration store.

**See Also**

**Functions**
commit

**Properties**
MasterLocation, ProcessLocation
**Purpose**  
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 string.

**Characteristics**  
Usage: VISA object  
Read only: Always  
Data type: String

**Values**  
The default value is an empty string.

**Remarks**  
You set the alias for a resource name using vendor-supplied tools. You do not set an alias in MATLAB. When you create the VISA object in MATLAB, 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 MATLAB to access your resources.

**See Also**  
**Functions**  
visa

**Properties**  
RsrcName
**BaudRate**

**Purpose**
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.

**See Also**

- Properties
  - DataBits, Parity, StopBits
**BoardIndex**

**Purpose**
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.

```matlab
vg = visa('agilent','GPIB4::1::8::INSTR');
```

The `BoardIndex`, `Name`, and `RsrcName` properties reflect the GPIB board index number.

```matlab
get(vg,{'BoardIndex','Name','RsrcName'})
ans =
[4] 'VISA-GPIB4-1-8' 'GPIB4::1::8::INSTR'
```
See Also

Functions
fclose, gpib, visa

Properties
Name, RsrcName, Status
**Purpose**

Specify M-file callback function to execute when break-interrupt event occurs.

**Description**

You configure `BreakInterruptFcn` to execute an M-file 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

To learn how to create a callback function, refer to “Creating and Executing Callback Functions” on page 4-32.

**Characteristics**

- **Usage**: Serial port
- **Read only**: Never
- **Data type**: Callback function

**Values**

The default value is an empty string.

**See Also**

- **Functions**
  - `record`
Properties

RecordStatus
**Purpose**  
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.

```matlab  
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.

```matlab  
fprintf(g,'*STB?')  
g.BusManagementStatus
```
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.

```plaintext
g.busmanagementstatus
ans =
    Attention: 'on'
    InterfaceClear: 'off'
    RemoteEnable: 'on'
    ServiceRequest: 'off'
    EndOrIdentify: 'off'
```

ATN is asserted because a multiline response was read from the scope.

```plaintext
close(g)
delete(g)
clear g
```
ByteOrder

Purpose
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.

Characteristics

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Usage</th>
<th>Any instrument object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td></td>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

<table>
<thead>
<tr>
<th>{littleEndian}</th>
<th>The byte order of the instrument is little-endian.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigEndian</td>
<td>The byte order of the instrument is big-endian.</td>
</tr>
</tbody>
</table>

See Also
Properties

Status
BytesAvailable

**Purpose**
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 line only after the input buffer is empty. Therefore, the BytesAvailable value is always 0. To learn how to read data asynchronously, refer to “Synchronous Versus Asynchronous Read Operations” on page 3-23.

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
**Purpose**
Specify M-file callback function to execute when specified number of bytes are available in input buffer, or terminator is read.

**Description**
You configure `BytesAvailableFcn` to execute an M-file 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

To learn how to create a callback function, refer to “Creating and Executing Callback Functions” on page 4-32.

**Characteristics**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Any instrument object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>Callback function</td>
</tr>
</tbody>
</table>

**Values**
The default value is an empty string.

**Examples**
Create the serial port object `s` for a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1.

```
s = serial('COM1');
```
Configure `s` to execute the M-file callback function `instrcallback` when 40 bytes are available in the input buffer.

```matlab
s.BytesAvailableFcnCount = 40;
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcn = @instrcallback;
```

Connect `s` to the oscilloscope.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
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.

```matlab
s.BytesAvailable
ans =
    102
```

**See Also**

**Functions**

record

**Properties**

BytesAvailableFcnCount, BytesAvailableFcnMode, EOSCharCode, RecordStatus, Terminator, TransferStatus
**Purpose**
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 M-file 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**

**Purpose**
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 M-file 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**: String

**Values**
- **Serial, TCPIP, UDP, and VISA-Serial**
  - `{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**

{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**

**Purpose**
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 line only after the output buffer is empty. Therefore, the BytesToOutput value is always 0. To learn how to write data asynchronously, Refer to “Synchronous Versus Asynchronous Write Operations” on page 3-17.

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
**Purpose**
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.

```matlab
v = visa('agilent','GPIB-VXI0::32::INSTR');
```

The `ChassisIndex`, `Name`, and `RsrcName` properties reflect the VXI chassis index number.

```matlab
get(v,{'ChassisIndex','Name','RsrcName'})
ans =
 0 'VISA-GPIB-VXI0-32' 'GPIB-VXI0::32::INSTR'
```

**See Also**
- **Functions**
  - `fclose`, `visa`
Properties

Name, RsncName, Status
**Purpose**

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**

- `{8}` Compare all eight EOS bits.
- `7` Compare the lower seven EOS bits.

**See Also**

**Properties**

EOSCharCode, EOSMode
### ConfirmationFcn

#### Purpose
Specify M-file callback function to execute when confirmation event occurs.

#### Description
You configure `ConfirmationFcn` to execute an M-file 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

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Usage</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>Callback</td>
<td></td>
</tr>
</tbody>
</table>

#### Values
The default value is an empty string.

#### See Also
**Functions**

connect
**DataBits**

**Purpose**
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**

**Purpose**  
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 string 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**: String

**Values**

The default value is ''. 

**See Also**

**Functions**

udp

**Properties**

DatagramPort, RemotePort
**Purpose**
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**

**Purpose**
Specify M-file callback function to execute when datagram is received

**Description**
You configure `DatagramReceivedFcn` to execute an M-file 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

To learn how to create a callback function, refer to “Creating and Executing Callback Functions” on page 4-32

**Characteristics**

<table>
<thead>
<tr>
<th>Usage</th>
<th>UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>Callback</td>
</tr>
</tbody>
</table>

**Values**
The default value is ' '.

**See Also**

**Functions**
readasync, udp

**Properties**

DatagramAddress, DatagramPort, ReadAsyncMode
**Purpose**
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**: String

**Values**

- **{on}**
  - The read operation terminates when a datagram is read.

- **off**
  - The read operation spans datagram boundaries.

**See Also**

**Functions**

fread, fscanf, udp
DataTerminalReady

**Purpose**
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 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 “The Control Pins” on page 6-7.

**Characteristics**
- Usage: Serial port, VISA-serial
- Read only: Never
- Data type: String

**Values**
- {on}: The DTR pin is asserted.
- off: The DTR pin is unasserted.

**See Also**

**Properties**
- `FlowControl`, `PinStatus`
**Purpose**
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, IVI-C, or IVI-COM, the `DriverName` is the name of the VXIplug&play, IVI-C, or IVI-COM driver, respectively.

**Characteristics**
- **Usage**: Device
- **Read only**: Always
- **Data type**: String

**Values**
`DriverName` is defined at device object creation.

**See Also**
- **Properties**
  - `DriverType`
**DriverSessions**

**Purpose**
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
**Purpose**
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 VXIplug&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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Always</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**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**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Specify if EOI line is asserted at end of write operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Characteristics**

- **Usage**: GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI
- **Read only**: Never
- **Data type**: String

**Values**

- **{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
**Purpose**
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 given in the EOSMode property description.

**Characteristics**

<table>
<thead>
<tr>
<th>Usage</th>
<th>GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>ASCII value</td>
</tr>
</tbody>
</table>

**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
**Purpose**

Specify when EOS character is written or read

**Description**

For GPIB, VISA-GPIB, VISA-VXI, and VISA-GPIB-VXI 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
- **Read only**: Never
- **Data type**: String

**Values**

- {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.

```plaintext
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.

```plaintext
set(g,'EOSMode','read')
set(g,'EOSCharCode','E')
fprintf(g,'Volt?')
out = fscanf(g)
out = +2.00000
```
See Also

Properties

CompareBits, EOIMode, EOSCharCode
**Purpose**

Specify M-file callback function to execute when error event occurs

**Description**

You configure `ErrorFcn` to execute an M-file 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

To learn how to create a callback function, refer to “Creating and Executing Callback Functions” on page 4-32.

**Characteristics**

- **Usage**: Any instrument object
- **Read only**: Never
- **Data type**: Callback function

**Values**

The default value is an empty string.
## See Also Functions

record

### Properties

RecordStatus, Timeout
Purpose
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 “Using Control Pins” on page 6-30.

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 and a hardware connection is not detected, the fwrite and the fprintf functions will return an error message.

Note
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

<table>
<thead>
<tr>
<th>Usage</th>
<th>Serial port, VISA-serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>
### FlowControl

**Values**

- `{none}`: No flow control is used.
- `hardware`: Hardware flow control is used.
- `software`: Software flow control is used.

**See Also**

**Properties**

- `PinStatus`
- `RequestToSend`
HandshakeStatus

Purpose
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.
**Purpose**  
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
**Purpose**

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**

**Purpose**
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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Device Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Always</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**
The default value is defined at the device group object creation.

**See Also**

**Properties**

HwIndex
Purpose
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.

See Also
Functions
fclose, fgetl, fgets, fopen, fread, fscanf

Properties
Status
**Purpose**
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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Always</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**
InstrumentModel will be an empty string until the object is connected to the instrument with the connect function and the property value is queried with the get function.

**See Also**

**Functions**

connect, get
Interface

Purpose

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. If DriverType is IVI-COM, Interface is the handle to the IVI driver’s default interface.

Characteristics

<table>
<thead>
<tr>
<th>Usage</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Always</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

Values

Interface is defined at device object creation.

See Also

Properties

DriverType, LogicalName, RsrcName
**InterfaceIndex**

**Purpose**
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`
Purpose
Specify M-file callback function to execute when interrupt event occurs

Description
You configure `InterruptFcn` to execute an M-file 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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>VISA-VXI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**
The default value is an empty string.

**See Also**

- **Functions**
  record

- **Properties**
  RecordStatus
**Purpose**
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**: String

**See Also**
- **Functions**: `fclose`
- **Properties**: `Name`, `RsrcName`
**Purpose**
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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>TCPIP, UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>While open</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**
The default value is an empty string.

**See Also**

**Functions**
`fclose, fopen, tcpip, udp`

**Properties**
`LocalPort, RemoteHost, Status`
**Purpose**
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**

**Purpose**
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`, MATLAB 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**: String

**Values**
- `{auto}`: Use any free local port.
- `manual`: Use the specified local port value.

**See Also**
- **Functions**
  - `fclose`, `fopen`, `tcpip`, `udp`

**Properties**
- `LocalHost`, `LocalPort`, `Status`
**Purpose**  
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.

```matlab  
nv = visa('agilent','VXI4::1::INSTR');  
nv.LogicalAddress  
an =  
1  
```

**See Also**  
- **Functions**: `fclose`, `visa`  

**Properties**  
- `Name`, `RsrcName`, `Status`
**Purpose**

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 or IVI-COM, the `LogicalName` is the `LogicalName` associated with the IVI-C or IVI-COM driver.

**Characteristics**

Usage: Device

Read only: Always

Data type: String

**Values**

`LogicalName` is defined at device object creation.

**See Also**

**Properties**

`DriverType`, `Interface`, `RsncName`
**LogicalNames**

**Purpose**
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
**Purpose** Specify manufacturer ID of USB instrument

**Description** You configure ManufacturerID to be the manufacturer ID of the USB instrument.

The Name and RsrgName 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** String

**See Also**

**Functions**

fclose

**Properties**

Name, RsrgName
**MappedMemoryBase**

**Purpose**

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 string 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: String

**Values**

The default value is 0H.

**Examples**

Create the VISA-VXI object vv associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

```matlab
vv = visa('agilent','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.

```matlab
memmap(vv,'A16',0,16)
vv.MappedMemoryBase
ans =
16737610H
```
See Also

Functions
memmap, memunmap

Properties
MappedMemorySize
**MappedMemorySize**

**Purpose**
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 an Agilent E1432A digitizer with logical address 130.

```matlab
vv = visa('agilent','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.

```matlab
memmap(vv,'A16',0,16)
vv.MappedMemorySize
ans =
    16
```

**See Also**
- **Functions**
  - `memmap`, `memunmap`

**Properties**
- `MappedMemoryBase`
**Purpose**
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**: String

**Values**
The default value is set at IVI installation.

**See Also**
Properties
ActualLocation, ProcessLocation
MemoryBase

**Purpose**  
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 string 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: String

**Values**  
The default value is 0H.

**Examples**  
Create the VISA-VXI object vv associated with a VXI chassis with index 0, and an Agilent E1432A digitizer with logical address 130.

```matlab
vv = visa('agilent','VXI0::130::INSTR');
fopen(vv)
```

The MemorySpace property indicates that the A24 memory space is supported.

```matlab
vv.MemorySpace
ans =
A16/A24```
The base address of the A24 space is

```matlab
vv.MemoryBase
ans =
'200000H'
```

**See Also**

**Properties**

MemorySpace
**MemoryIncrement**

**Purpose**
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**: String

**Values**
- **{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.

```matlab
v = visa('ni','VXI0::8::INSTR');
 fopen(v)
```

Configure the hardware for a FIFO read and write operation.

```matlab
set(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.

```matlab
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.

\[
\text{memread}(v, 16, 'uint32')
\]
\[\text{ans} = 2000\]

Read two values starting at offset 16. Note that both values are read at offset 16.

\[
\text{memread}(v, 16, 'uint32', 'A16', 2);
\]
\[\text{ans} =
\begin{align*}
2000 \\
2000
\end{align*}\]

Configure the hardware for a block read and write operation.

\[
\text{set}(v, 'MemoryIncrement', 'block')
\]

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.

\[
\text{memwrite}(v, [1984 2000], 16, 'uint32', 'A16')
\]

Read the value at offset 16. The value returned is the first value written with the `memwrite` function.

\[
\text{memread}(v, 16, 'uint32')
\]
\[\text{ans} = 1984\]

Read two values starting at offset 16. The first value is read at offset 16 and the second value is read at offset 20.

\[
\text{memread}(v, 16, 'uint32', 'A16', 2);
\]
\[\text{ans} =
\begin{align*}
1984 \\
2000
\end{align*}\]
MemoryIncrement

See Also

Functions

mempeek, mempoke, memread, memwrite
**Purpose**
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 an Agilent E1432A digitizer with logical address 130.

```python
vv = visa('agilent','VXI0::130::INSTR');
fclose(vv)
```

The MemorySpace property indicates that the A24 memory space is supported.

```python
vv.MemorySpace
ans =
A16/A24
```

The size of the A24 space is

```python
vv.MemorySize
ans =
262144
```
## MemorySize

### See Also

### Properties

- MemorySpace
MemorySpace

**Purpose** 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**: String

**Values**

- `{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 an Agilent E1432A digitizer with logical address 130.

```matlab
vv = visa('agilent','VXI0::130::INSTR');
fopen(vv)
```
MemorySpace

Return the memory space supported by the instrument.

```matlab
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
Purpose  Specify model code of USB instrument

Description  You configure ModelCode to be the model code of the USB instrument.
The Name and RsncName 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  String

See Also  
Functions  fclose
Properties  Name, RsncName
Name

**Purpose**

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.

<table>
<thead>
<tr>
<th>Instrument Object</th>
<th>Default Value of Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIB</td>
<td>GPIB and BoardIndex-PrimaryAddress-SecondaryAddress</td>
</tr>
<tr>
<td>serial port</td>
<td>Serial and Port</td>
</tr>
<tr>
<td>TCPIP</td>
<td>TCPIP and RemoteHost</td>
</tr>
<tr>
<td>UDP</td>
<td>UDP and RemoteHost</td>
</tr>
<tr>
<td>VISA-serial</td>
<td>VISA-Serial and Port</td>
</tr>
<tr>
<td>VISA-GPIB</td>
<td>VISA-GPIB and BoardIndex-PrimaryAddress-SecondaryAddress</td>
</tr>
<tr>
<td>VISA-VXI</td>
<td>VISA-VXI and ChassisIndex-LogicalAddress</td>
</tr>
<tr>
<td>VISA-GPIB-VXI</td>
<td>VISA-GPIB-VXI and ChassisIndex-LogicalAddress</td>
</tr>
<tr>
<td>VISA-TCPIP</td>
<td>VISA-TCPIP and BoardIndex-RemoteHost-LANName</td>
</tr>
<tr>
<td>VISA-RSIB</td>
<td>VISA-RSIB and RemoteHost</td>
</tr>
<tr>
<td>VISA-USB</td>
<td>VISA-USB and BoardIndex-ManufacturerID-ModelCode-SerialNumber-InterfaceIndex</td>
</tr>
</tbody>
</table>

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**: String

**Values**
Name is automatically defined at object creation time. The value of Name depends on the specific instrument object you create.
Name (iviconfigurationstore)

**Purpose**
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**: String
**Purpose**

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 M-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**: String

**Values**

The default value is `on`.

- `{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`:

```matlab
    g = gpib('keithley',0,2,'ObjectVisibility','off');
    instrfind
    ans =
        []
```

However, since the object is in the workspace (g), you can access it.

```matlab
    get(g,'ObjectVisibility')
    ans =
```
off

See Also

Functions

instrfind, instrfindall, instrreset
OutputBufferSize

**Purpose**
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.

**See Also**
- **Functions**
  - `fprintf`, `fwrite`

**Properties**
- `Status`
OutputEmptyFcn

Purpose
Specify M-file callback function to execute when output buffer is empty

Description
You configure `OutputEmptyFcn` to execute an M-file 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

To learn how to create a callback function, refer to “Creating and Executing Callback Functions” on page 4-32.

Characteristics
Usage
Any instrument object

Read
Never

only

Data type
Callback function

Values
The default value is an empty string.

See Also
Functions
`record`
Properties

RecordStatus
# Parent

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Parent (device object) of device group object</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>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 <code>Parent</code>. This copy can be treated like any other device object. For example, you can configure property values, connect to the instrument, and so on.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Usage</th>
<th>Device group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Always</td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>Device object</td>
<td></td>
</tr>
</tbody>
</table>

| Values | Parent is defined at device object creation. |
**Purpose**
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 “The Parity Bit” on page 6-12.

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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Serial port, VISA-serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

| {none} | No parity checking |
| odd | Odd parity checking |
| even | Even parity checking |
| mark | Mark parity checking |
| space | Space parity checking |
Parity

See Also

Properties

DataBits, StopBits
**Purpose**
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 “The Control Pins” on page 6-7 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 M-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 “Example: Connecting Two Modems” on page 6-31 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**

**Purpose** Specify M-file callback function to execute when CD, CTS, DSR, or RI pin changes state

**Description** You configure `PinStatusFcn` to execute an M-file 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

To learn how to create a callback function, refer to “Creating and Executing Callback Functions” on page 4-32.

**Characteristics**

- **Usage**: Serial port
- **Read only**: Never
- **Data type**: Callback function

**Values** The default value is an empty string.
See Also

Functions
record

Properties
PinStatus, RecordStatus
**Purpose**
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**: String

**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.

```matlab
s = serial('COM1')
vs = visa('ni','ASRL1::INSTR')
```

The `Port` property values are given below.

```matlab
get([s vs],'Port')
ans =
    'COM1'
    'ASRL1'
```

**See Also**
**Functions**
- `fclose`
- `serial`
- `visa`
# Port Properties

- Name
- RsncName
- Status
Purpose

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.

```matlab
vg = visa('agilent','GPIB0::1::8::INSTR');
vg.PrimaryAddress
ans =
    1
```

See Also

Functions

close, gpib, visa

Properties

Name, RsrqName, Status
**Purpose**  
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: String

**Values**  
The default value is an empty string.

**See Also**  
Functions  
iviconfigurationstore

**Properties**  
ActualLocation, MasterLocation
Published APIs

**Purpose**
Array of published APIs in IVI configuration store

**Description**
Published APIs 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
Purpose
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

<table>
<thead>
<tr>
<th>Usage</th>
<th>Serial port, TCPIP, UDP, VISA-serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>
**Values**

{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`
**Purpose**

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 “Example: Recording Information to Disk” on page 12-10.

**Characteristics**

Usage: Any instrument object  
Read only: Never  
Data type: String

**Values**

- **{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**

**Purpose**
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 12-8 for a description of these rules.

**Characteristics**
- **Usage**: Any instrument object
- **Read only**: While recording
- **Data type**: String

**Values**
- `{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` associated with the serial port COM1.

```matlab
s = serial('COM1');
fopen(s)
```
Specify the record filename with the `RecordName` property, configure `RecordMode` to `index`, and initiate recording.

```matlab
s.RecordName = 'myrecord.txt';
s.RecordMode = 'index';
record(s)
```

The record filename is automatically updated with an indexed filename after recording is turned off.

```matlab
record(s,'off')
s.RecordName
ans =
    myrecord01.txt
```

Disconnect `s` from the instrument, and remove `s` from memory and from the MATLAB workspace.

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

**See Also**

**Functions**

- `record`

**Properties**

- `RecordDetail`
- `RecordName`
- `RecordStatus`
RecordName

**Purpose**

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.

MATLAB supports any filename supported by your operating system. However, if you access the file through MATLAB, 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 line, 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 12-8 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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Any instrument object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>While recording</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

The default record file name is *record.txt*.

**See Also**

**Functions**

record

**Properties**

RecordDetail, RecordMode, RecordStatus
RecordStatus

**Purpose**
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**: String

**Values**
- **{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

Purpose
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
String

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

**Purpose**
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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>TCPIP, UDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>While open (TCPIP), never (UDP)</td>
</tr>
<tr>
<td>Data type</td>
<td>Double</td>
</tr>
</tbody>
</table>

**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**

**Purpose**
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 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 “The Control Pins” on page 6-7.

**Characteristics**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Serial port, VISA-serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{on}</td>
<td>The RTS pin is asserted.</td>
</tr>
<tr>
<td>{off}</td>
<td>The RTS pin is unasserted.</td>
</tr>
</tbody>
</table>

**See Also**

**Properties**

`FlowControl`, `PinStatus`
**Purpose**  
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**: String
**Purpose**
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.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Resource Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>VXI</td>
<td>VXI[chassis]::VXI_logical_address::INSTR</td>
</tr>
<tr>
<td>GPIB-VXI</td>
<td>GPIB-VXI[chassis]::VXI_logical_address::INSTR</td>
</tr>
<tr>
<td>GPIB</td>
<td>GPIB[board]::primary_address[::secondary_address]::INSTR</td>
</tr>
<tr>
<td>TCPIP</td>
<td>TCPIP[board]::remote_host[::lan_device_name]::INSTR</td>
</tr>
<tr>
<td>RSIB</td>
<td>RSIB::remote_host::INSTR</td>
</tr>
<tr>
<td>Serial</td>
<td>ASRL[port_number]::INSTR</td>
</tr>
<tr>
<td>USB</td>
<td>USB[board]::manid::model_code::serial_No[::interface_No]::INSTR</td>
</tr>
</tbody>
</table>

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**: String

**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.

\[ vg = \text{visa}('ni','GPIB0::1::INSTR'); \]

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.

\[ vv = \text{visa}('agilent','VXI0::130::INSTR'); \]

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.

\[ vgv = \text{visa}('agilent','GPIB-VXI0::80::INSTR'); \]

To create a VISA-serial object associated with the COM1 serial port, you supply the following resource name to the visa function.

\[ vs = \text{visa}('ni','ASRL1::INSTR'); \]

See Also

Functions

visa

Properties

BoardIndex, ChassisIndex, InterfaceIndex, LANName, LogicalAddress, ManufacturerID, ModelCode, Port, PrimaryAddress, RemoteHost, SecondaryAddress, SerialNumber
**SecondaryAddress**

**Purpose**
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.

```matlab
vg = visa('agilent','GPIB0::1::8::INSTR');
vg.SecondaryAddress
ans =
   8
```

See Also
Functions
fclose, gpib, visa

Properties
Name, RsncName, Status
SerialNumber

**Purpose**
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**: String

**See Also**

- **Functions**
  - `fclose`

- **Properties**
  - `Name`, `RsrcName`
ServerDescription

**Purpose**  
IVI configuration store server description

**Description**  
ServerDescription contains the description of the IVI configuration store server. This is not user-configurable.

**Characteristics**  

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage</td>
<td>IVI configuration store object</td>
</tr>
<tr>
<td>Read only</td>
<td>Always</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>
Sessions

**Purpose**  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
**Purpose**  
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

**Purpose**
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
<table>
<thead>
<tr>
<th><strong>Purpose</strong></th>
<th>IVI configuration server specification version that this server revision complies with</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>SpecificationVersion identifies the specification version of the IVI configuration store server. This is not user-configurable.</td>
</tr>
<tr>
<td><strong>Characteristics</strong></td>
<td><strong>Usage</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Read only</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Data type</strong></td>
</tr>
</tbody>
</table>
**Status**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Status of whether object is connected to instrument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
<tr>
<td>Characteristics</td>
<td>Usage</td>
</tr>
<tr>
<td></td>
<td>Read only</td>
</tr>
<tr>
<td></td>
<td>Data type</td>
</tr>
<tr>
<td>Values</td>
<td>{closed}</td>
</tr>
<tr>
<td></td>
<td>open</td>
</tr>
<tr>
<td>See Also</td>
<td>Functions</td>
</tr>
</tbody>
</table>
**Purpose**

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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Serial port, VISA-serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>double</td>
</tr>
</tbody>
</table>

**Values**

**Serial Port**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>One stop bit is transmitted to indicate the end of a byte.</td>
</tr>
<tr>
<td>1.5</td>
<td>The stop bit is transferred for 150% of the normal time used to transfer one bit.</td>
</tr>
<tr>
<td>2</td>
<td>Two stop bits are transmitted to indicate the end of a byte.</td>
</tr>
</tbody>
</table>
StopBits

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

See Also

Properties

DataBits, Parity
**Purpose**
Specify label to associate with instrument object

**Description**
You configure Tag to be a string 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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Any instrument object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**
The default value is an empty string.

**Examples**
Suppose you create a serial port object associated with the serial port COM1.

```matlab
s = serial('COM1');
fopen(s);
```

You can assign `s` a unique label using Tag.

```matlab
set(s,'Tag','MySerialObj')
```

You can access `s` in the MATLAB workspace or in an M-file using the `instrfind` function and the Tag property value.

```matlab
s1 = instrfind('Tag','MySerialObj');
```

**See Also**

**Functions**

- `instrfind`
**Terminator**

**Purpose** 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.
See Also

Functions
fgetl, fgets, fprintf, fscanf

Properties
BytesAvailableFcnMode
**Purpose**
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 M-file callback function specified for `ErrorFcn` is executed.

**Characteristics**
- **Usage**: Any instrument object
- **Read only**: Never
- **Data type**: Double

**Values**
The default value is 10 seconds.

**See Also**
**Properties**
`ErrorFcn`
**Purpose**  Specify M-file callback function to execute when predefined period passes

**Description**  You configure `TimerFcn` to execute an M-file 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.

To learn how to create a callback function, refer to “Creating and Executing Callback Functions” on page 4-32.

**Characteristics**  Usage  Any instrument object

Read only  Never

Data type  Callback function

**Values**  The default value is an empty string.
See Also

Functions
fopen, record

Properties
RecordStatus, TimerPeriod
Purpose  
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  
Callback function

Values  
The default value is 1 second. The minimum value is 0.01 second.

See Also

Functions  
fopen

Properties  
TimerFcn
### TransferDelay

**Purpose**
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**: String

**Values**
- {on} Use the TCP segment transfer algorithm.
- off Do not use the TCP segment transfer algorithm.

**See Also**
- Functions:
  - tcpip
### Purpose
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 2-7.

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**: String

### Values
- **{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.
TransferStatus

See Also Functions
fprintf, fwrite, readasync

Properties
ReadAsyncMode
**Purpose**
Specify M-file callback function to execute when trigger event occurs

**Description**
You configure TriggerFcn to execute an M-file 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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>VISA-VXI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**
The default value is an empty string.

**See Also**

- **Functions**
  - record

- **Properties**
  - RecordStatus, TriggerLine, TriggerType
**Purpose**
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**

<table>
<thead>
<tr>
<th>Usage</th>
<th>VISA-VXI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>

**Values**
TriggerLine can be TTL0 through TTL7, ECL0, or ECL1. The default value is TTL0.

**See Also**

**Functions**
trigger

**Properties**
TriggerType
### Purpose
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**: String

### Values
- `{hardware}`: A hardware trigger is used.
- `software`: A software trigger is used.

### See Also
- **Functions**
  - `trigger`
- **Properties**
  - `TriggerLine`
Type

Purpose
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

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Usage</th>
<th>Any instrument object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Always</td>
<td></td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>

Values

- **gpib**: The object type is GPIB.
- **serial**: The object type is serial port.
- **tcip**: 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 associated with the serial port COM1. The value of the Type property is serial, which is the object class.

```matlab
s = serial('COM1');
s.Type
ans =
serial
```

See Also

Functions

instrfind, gpib, serial, tcpip, udp, visa
**UserData**

### Purpose
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 the `get` function or the dot notation.

### Characteristics

<table>
<thead>
<tr>
<th>Usage</th>
<th>Any instrument object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Never</td>
</tr>
<tr>
<td>Data type</td>
<td>Any type</td>
</tr>
</tbody>
</table>

### Values
The default value is an empty vector.

### Examples
Suppose you create the serial port object associated with the serial port COM1.

```matlab
s = serial('COM1');
```

You can associate data with `s` by storing it in `UserData`.

```matlab
coeff.a = 1.0;
coeff.b = -1.25;
s.UserData = coeff
```
**Purpose**
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. Refer to “Output Buffer and Data Flow” on page 3-14 for more information about bytes and values.

**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 associated with the serial port COM1.

```matlab
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.

```matlab
fprintf(s,'RS232?')
out = fscanf(s)
out =
  9600;0;0;NONE;LF
s.ValuesReceived
ans =
  17
```

**See Also**

- **Functions**
  - fopen

- **Properties**
  - BytesAvailable
**Purpose**
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. Refer to “Output Buffer and Data Flow” on page 3-14 for more information about bytes and values.

**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 associated with the serial port COM1.

```matlab
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.

```matlab
fprintf(s,'*IDN?')
s.ValuesSent
ans =
 6
```

**See Also**

**Functions**

fopen

**Properties**

BytesToOutput
**Purpose**  
IVI configuration server vendor

**Description**  
Vendor identifies the vendor of the IVI configuration server. This is not user-configurable.

**Characteristics**  
<table>
<thead>
<tr>
<th>Usage</th>
<th>IVI configuration store object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read only</td>
<td>Always</td>
</tr>
<tr>
<td>Data type</td>
<td>String</td>
</tr>
</tbody>
</table>
Blocks — Alphabetical List
Purpose

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.

Dialog Box

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.
Query Instrument

Board vendor
The vendor of the GPIB board used for the interface to the instrument. Your choices are Advantech, Agilent, Capital Equipment, Contec, ICS, IOTech, Keithley, Measurement Computing, 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 Instrument Control Toolbox. 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.

Model code
The model code of the VISA USB instrument.

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, TCP/IP, 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.

Timeout
Time ion 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 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 form 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 Instrument Block Parameters](image)

**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 (default)
- 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 the dialog box is first opened. To refresh the display with new values, restart MATLAB.

See Also
To Instrument, TCP/IP Receive, TCP/IP Send, UDP Receive, UDP Send
To Instrument

**Purpose**
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.

**Dialog Box**

**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.

**Board vendor**

The vendor of the GPIB board used for the interface to the instrument. Your choices are Advantech, Agilent, Capital
To Instrument


**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 Instrument Control Toolbox. 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.

**Model code**
The model code of the VISA USB instrument.

**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.

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 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 form 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)
To Instrument

- 16-bit integer
- 32-bit integer
- 8-bit unsigned integer (default)
- 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**.

---

**Note** Hardware information shown in the dialog box is determined and cached when the dialog box is first opened. To refresh the display with new values, restart MATLAB.

---

**See Also** Query Instrument, TCP/IP Receive, TCP/IP Send, UDP Receive, UDP Send
**Purpose**
receive data over TCP/IP from specified remote machine

**Library**
Instrument Control Toolbox

**Description**
The TCP/IP Receive block configures and opens an interface to a specified remote address using the TCP/IP 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 mode or nonblocking mode.

*Note* You must have a license for both Instrument Control Toolbox and Simulink 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 then the block will have one output port corresponding to the data it receives.

If you do not select blocking mode, the block will have two output ports, the **Data** port and the **Status** port.

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 nonblocking mode, the **Status** port indicates if the block has received new data.
TCP/IP Receive

Use the TCP/IP Receive block to read streaming data over a TCP/IP network, using the Instrument Control Toolbox functionality in Simulink.

**Dialog Box**

Use the Source Block Parameters dialog box to select your communication parameters.

![Source Block Parameters: TCP/IP Receive](image)

**Remote address**

Enter the IP address, name, or the Web server address of the machine from which you need to receive data. This field is empty by default.

**Port**

Enter the remote port on the remote machine you need to connect to. The default port value is 80. 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.

**Data size**
Specify the output data size, or the number of values that should be read at every simulation time step. The default size is $[1 \ 1]$.

**Data type**
Specify the output data type to receive from the block. You can select from the following values:

• single
• double
• int8
• uint8 (default)
• int16
• uint16
• int32
• uint32

**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. When 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 specifies the amount of data to receive as 4 (set in the Data size field of the TCP/IP Receive Block Parameters dialog box). Once 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: Simulink 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: Simulink 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 Simulink 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 TCP/IP Receive Block Parameters dialog box), a Simulink error occurs and the simulation is stopped.

**Nonblocking Mode**

Here the simulation is not blocked and runs continuously.

- At time step 1: Simulink 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: Simulink 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 Simulink 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 10 (seconds). This field is disabled if you have not selected Enable blocking mode.

**Block sample time**
Specify the sample time of the block during the simulation. This is the rate at which the block is executed during simulation. The default value is 0.01 (seconds).

**See Also**
Query Instrument, To Instrument, TCP/IP Send, UDP Receive, UDP Send
Purpose
Send data over TCP/IP to specified remote machine

Library
Instrument Control Toolbox

Description
The TCP/IP Send block sends data from your model to remote machines using the TCP/IP protocol. This data is sent at the end of the simulation or at fixed intervals during a simulation.

Note You must have a license for both Instrument Control Toolbox and Simulink to use this block.

The TCP/IP Send block has one input port. The size of the input port is dynamic, and is inherited from the driving block. This block has no output ports.

Dialog Box
Use the Sink Block Parameters dialog box to select your communication parameters.

![Sink Block Parameters: TCP/IP Send](image)
**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.

**Port**

Specify the remote port on the host you need to send the data to. The default port value is 80. 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.

**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. Once 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 speed outpaces the data acquisition.

- 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.
**Timeout**

Specify the amount of time that the model will wait when data is sent during each simulation time step. The default value is 10 (seconds). This field is unavailable if you have not enabled blocking mode.

**See Also**

Query Instrument, To Instrument, TCP/IP Receive, UDP Receive, UDP Send
**Purpose**  
Receive data over UDP network from specified remote machine

**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 must have a license for both Instrument Control Toolbox and Simulink 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.

If you do not select blocking mode, the block will have two output ports, the **Data** port and the **Status** port.

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.
Use the Source Block Parameters dialog box to select your acquisition mode and to set other configuration options.

**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 empty by default.

**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.

**Local port**
Specify the port to bind on the local machine. The default value is -1, which automatically binds to an available port.

**Verify address and port connectivity ...**
Click this button to
UDP Receive

- Check if the specified remote address is correct.
- Establish connection with the specified remote address and port.

**Data size**
Specify the output data size, or the number of values that should be read at every simulation time step. The default size is $[1\ 1]$.

**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 specifies the amount of data to receive as 4 (set in the **Data size** field of the UDP Receive Block Parameters dialog box). Once the simulation starts, the data is acquired asynchronously in a FIFO buffer.
The blocking mode simulation occurs like this:

- At time step 1: Simulink 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: Simulink 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 Simulink 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 error occurs and the simulation is stopped.
Here the simulation is not blocked and runs continuously.

- At time step 1: Simulink 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: Simulink 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 Simulink 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 $\infty$ (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).

**Data type**
Specify the output data type to receive from the block. You can select from the following values:

- single
- double
- int8
- uint8 (default)
- int16
- uint16
- int32
- uint32

**See Also**
Query Instrument, To Instrument, TCP/IP Receive, TCP/IP Send, UDP Send
UDP Send

**Purpose**
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 must have a license for both Instrument Control Toolbox and Simulink 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.
Use the Sink Block Parameters dialog box to select your acquisition mode and to set other configuration options.

**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.

**Port**
Specify the remote port on the host you need to send the data to. The default port value is 80. Valid port values are 1 to 65535.

**Local port**
Specify the port to bind on the local machine. The default value is -1, which automatically binds to an available port.
UDP Send

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.

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. Once 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 speed outpaces the data acquisition.

- 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.

Here the simulation is nonblocking and occurs faster than the data initiation. Data is initiated at time step T1, . When the second data initiation occurs at T2, the to the specified remote address and port. The simulation resumes and B2 is sent. When the next data initiation at T3 occurs, the simulation is blocked until B2 is sent.

**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, To Instrument, TCP/IP Receive, TCP/IP Send, UDP Receive
Vendor Driver Requirements and Limitations

This appendix describes the requirements and limitations for the vendor GPIB and VISA drivers supported by Instrument Control Toolbox.

Driver Requirements (p. A-2)  Required versions of vendors’ drivers.

GPIB Driver Limitations by Vendor (p. A-3)  Known limitations with supported GPIB drivers.

VISA Driver Limitations (p. A-7)  Known limitations with supported VISA drivers.

**Note** The limitations described in this appendix are restricted to the limitations directly associated with using Instrument Control Toolbox.
**Driver Requirements**

You can use Instrument Control Toolbox with the following GPIB and VISA drivers.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Vendor</th>
<th>Minimum Driver Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPIB</td>
<td>Advantech</td>
<td>Advantech version 1.0</td>
</tr>
<tr>
<td></td>
<td>Agilent</td>
<td>SICL version H.012.02.00</td>
</tr>
<tr>
<td></td>
<td>Capital Equipment Corporation</td>
<td>CEC-488 version 5.08</td>
</tr>
<tr>
<td></td>
<td>CONTEC</td>
<td>API-GPLV version 1.33</td>
</tr>
<tr>
<td></td>
<td>ICS Electronics</td>
<td>ICS 488.2 version 2.0.3.15</td>
</tr>
<tr>
<td></td>
<td>IOTech</td>
<td>IOTech version 1.0</td>
</tr>
<tr>
<td></td>
<td>Keithley</td>
<td>Keithley-488 version 5.08</td>
</tr>
<tr>
<td></td>
<td>Measurement Computing Company</td>
<td>GPIB-32 version 2.12</td>
</tr>
<tr>
<td></td>
<td>National Instruments</td>
<td>NI-488 version 1.6</td>
</tr>
<tr>
<td>VISA</td>
<td>Agilent</td>
<td>Agilent IO Libraries version M.01.01.04</td>
</tr>
<tr>
<td></td>
<td>National Instruments</td>
<td>NI-VISA version 3.0</td>
</tr>
<tr>
<td></td>
<td>Tektronix</td>
<td>Tektronix VISA version 2.0</td>
</tr>
</tbody>
</table>

See the following sections for a description of

- “GPIB Driver Limitations by Vendor” on page A-3
- “VISA Driver Limitations” on page A-7
GPIB Driver Limitations by Vendor

In this section...

“Advantech” on page A-3
“Agilent Technologies” on page A-3
“Capital Equipment Corporation” on page A-4
“ICS Electronics” on page A-4
“IO Tech” on page A-5
“Keithley” on page A-5
“Measurement Computing Corporation” on page A-6

Advantech
The Advantech GPIB driver has these limitations:

- Asynchronous read and write operations are not supported. Therefore, Advantech GPIB objects do not support the following toolbox functionality:
  - The readasync function
  - The async flag for the fprintf and fwrite functions
  - BytesAvailableAction and OutputEmptyAction properties

Agilent Technologies
The Agilent GPIB driver has these limitations:

- Asynchronous read and write operations are not supported. Therefore, Agilent GPIB objects do not support the following toolbox functionality:
  - The readasync function
  - The async flag for the fprintf and fwrite functions
  - BytesAvailableAction and OutputEmptyAction properties
- The End Or Identify (EOI) line is not asserted when the End-Of-String (EOS) character is written to the hardware. Therefore, when the EOSMode property is configured to write and the EOIMode property is configured to
on, the EOI line is not asserted when the EOSCharCode property value is written to the hardware.

- All eight bits are used for the EOS comparison. Therefore, the only value supported by the CompareBits property is 8.

- 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 will time out.

**Capital Equipment Corporation**

The Capital Equipment Corporation (CEC) GPIB driver has these limitations:

- Asynchronous read operations are not supported. Therefore, CEC GPIB objects do not support the following toolbox functionality:
  - The readasync function
  - The async flag for the fprintf and fwrite functions
  - The BytesAvailableAction and OutputEmptyAction properties

- The Handshake and Bus Management line values are not provided. The BusManagementStatus and HandshakeStatus properties always return the line values as on.

- The EOI line is not asserted when the EOS character is written to the hardware. Therefore, when the EOSMode property is configured to write and the EOIMode property is configured to on, the EOI line is not asserted when the EOSCharCode property value is written to the hardware.

- All eight bits are used for the EOS comparison. Therefore, the only value supported by the CompareBits property is 8.

- You should not simultaneously use a GPIB controller address of 0 and an instrument primary address of 0.

**ICS Electronics**

The ICS Electronics GPIB adaptor only works with their USB GPIB controller. However, you can still use their other controllers (PCI/PCMCIA) by using the mcc adaptor. See Solution 1-1BFZ2 on the MathWorks support Web site.
**IOTech**

The IOTech GPIB driver has these limitations:

- Asynchronous read and write operations are not supported. Therefore, IOTech GPIB objects do not support the following toolbox functionality:
  - The `readasync` function
  - The `async` flag for the `fprintf` and `fwrite` functions
  - The `BytesAvailableAction` and `OutputEmptyAction` properties.
- Incorrect values are returned for the REN and IFC bus management lines. The `BusManagementStatus` property always returns a value of `on` for the `RemoteEnable` and the `InterfaceClear` fields.
- The EOI line is not asserted when the EOS character is written to the hardware. Therefore, when the `EOSMode` property is configured to `write` and the `EOIMode` property is configured to `on`, the EOI line will not be asserted when the `EOSCharCode` property value is written to the hardware.

**Keithley**

The Keithley GPIB driver has these limitations:

- Asynchronous read and write operations are not supported. Therefore, Keithley GPIB objects do not support the following toolbox functionality:
  - The `readasync` function
  - The `async` flag for the `fprintf` and `fwrite` functions
  - The `BytesAvailableAction` and `OutputEmptyAction` properties
- The Handshake and Bus Management line values are not provided. The `BusManagementStatus` and `HandshakeStatus` properties always return the line value as `on`.
- The EOI line is not asserted when the EOS character is written to the hardware. Therefore, when the `EOSMode` property is configured to `write` and the `EOIMode` property is configured to `on`, the EOI line will not be asserted when the `EOSCharCode` property value is written to the hardware.
- All eight bits are used for the EOS comparison. Therefore, the only value supported by the `CompareBits` property is 8.
• You should not simultaneously use a GPIB controller address of 0 and an instrument primary address of 0.

**Measurement Computing Corporation**

The Measurement Computing Corporation GPIB driver does not support asynchronous notification for the completion of read and write operations. Therefore, Measurement Computing Corporation GPIB objects do not support the following toolbox functionality:

• The `readasync` function
• The `async` flag for the `fprintf` and `fwrite` functions
• The `BytesAvailableAction` and `OutputEmptyAction` properties
VISA Driver Limitations

Agilent Technologies
The Agilent VISA driver has these known limitations:

- Asynchronous read and write operations for GPIB, GPIB-VXI and VXI VISA objects are not supported. Therefore, the following toolbox functionality is not supported for these objects:
  - The readasync function
  - The BytesAvailableAction and OutputEmptyAction properties
- All eight bits are used for the EOS comparison. Therefore, the only value supported by the CompareBits property is 8.

National Instruments
The National Instruments VISA driver uses all eight bits for the EOS comparison. Therefore, the only value that the CompareBits property supports is 8.x
Bibliography


A
A16/A24/A32 memory space 5-15
active state, serial port 6-6
ActualLocation property 20-2
adaptors 1-8
  finding with instrhwinfo 1-13
add function 18-2
address configuration
  GPIB object 4-18
  VISA-GPIB object 5-7
  VISA-GPIB-VXI object 5-25
  VISA-RSIB object 5-41
  VISA-TCPIP object 5-37
  VISA-USB object 5-33
  VISA-VXI object 5-12
Agilent Technologies
  adaptors 1-8
  E1432A registers, example of 5-17
Alias property 20-3
array, instrument object
  creating 3-3
  example of 4-40
ASCII
  control characters 6-34
  read operations 4-20
    versus binary 3-21
  serial data 6-9
  write operations 4-20
    versus binary 3-15
asynchronous
  read operations 3-23
    readsasync example 4-23
    readsasync example using events 4-34
    ReadAsyncMode, example of 6-20
  serial port 6-10
  write operations 3-17
ATN line 4-7
  serial poll 4-39

B
base properties 2-5
  list for interface objects 19-2
BaudRate property 20-4
binary
  floating-point arithmetic standard 12-8
  read operations 4-22
    versus text 3-21
  write operations 3-15
binblockread function 18-6
binblockwrite function 18-9
Block Library 16-3
blocks
  Query Instrument 21-2
  TCP/IP Receive 21-17
  TCP/IP Send 21-23
  To Instrument 21-10
  UDP Receive 21-28
  UDP Send 21-34
  using the Instrument Control Toolbox block library 16-1
BoardIndex
  GPIB object 4-18
  VISA-GPIB object 5-8
  VISA-GPIB-VXI object 5-26
BoardIndex property 20-6
break-interrupt event
  datagram-received 7-21
  serial port 6-25
BreakInterruptFcn property 20-8
buffer
  clearing hardware 4-27
  input 3-20
  output 3-14
  values versus bytes 3-14
bus and connector, GPIB 4-4
BusManagementStatus
  GPIB interface management lines 4-8
BusManagementStatus property 20-10
  example of 20-10
Index

ByteOrder property 20-12
bytes versus values 3-14
bytes-available event 4-29
   example 4-34
BytesAvailable
   example of 3-24
BytesAvailable property 20-13
BytesAvailableFcn property 20-14
BytesAvailableFcnCount property 20-17
BytesAvailableFcnMode property 20-18
BytesToOutput
   output buffer 3-14
BytesToOutput property 20-20

C
   callback functions
      creating 4-32
      enabling after they error 4-33
      executing 4-32
      instrcallback, example 4-28
   callback properties
      GPIB object 4-29
      saving property values to a MAT-file 12-2
      serial port object 6-25
      TCP/IP object 7-21
      UDP object 7-21
CD pin 6-8
ChassisIndex property 20-21
   clear
      cleaning up the MATLAB environment 3-25
   clear function 18-12
   clrdevice
      example of 4-27
   clrdevice function 18-13
   commit function 18-14
   CompareBits property 20-23
   configuring property values 3-9
   ConfirmationFcn property 20-24
   connect function 18-15
   connecting to the instrument 2-4
      interface object 3-5
      constructor 3-2
      finding with instrhwinfo 1-13
   control characters 6-34
   control pins 6-7
      using 6-30
   controllers, GPIB 4-4
   creation function 3-2
   CTS pin 6-8

D
   data bits 6-11
   data format
      serial port 6-9
   data lines 4-7
   DataBits property 20-25
   DatagramAddress property 20-26
   DatagramPort property 20-27
   DatagramReceivedFcn property 20-28
   DatagramTerminateMode property 20-29
   DataTerminalReady property 20-30
   DAV line 4-8
   DCE 6-3
   dec2bin 12-9
   default property values 3-10
   delete
      cleaning up the MATLAB environment 3-25
   delete function 18-17
   demos 1-9
   device groups 8-14
   device objects 8-1
      compared to interface objects 2-3
      connecting 8-7
      control commands 8-12
      creating 8-5
      device groups 8-14
      methods 8-10
   devicereset function 18-19
DIO lines 4-7
disconnect function 18-20
disconnecting from the instrument 3-25
disp function 18-22
display summary
  GPIB object 4-16
  serial port object 6-17
  TCP/IP object 7-5
  UDP object 7-11
  VISA-GPIB object 5-7
  VISA-GPIB-VXI object 5-24
  VISA-RSIB object 5-40
  VISA-serial object 5-28
  VISA-TCPIP object 5-36
  VISA-USB object 5-32
  VISA-VXI object 5-11
documentation examples 1-9
dot notation
  configuring property values 3-9
  returning property values 3-9
  saving property values to an M-file 12-2
DriverName property 20-31
DriverSessions property 20-32
DriverType property 20-33
DSR pin 6-8
DTE 6-3
DTR pin 6-8

E
echotcpip function 18-23
echoudp function 18-24
enable registers 4-10
EOI line 4-7
  example of 4-25
EOIMode
  example 4-25
EOIMode property 20-34
EOS character 4-25
  EOSCharCode 20-35
  example 4-25
EOSCharCode property 20-35
EOSMode
  example 4-25
EOSMode property 20-36
error event 4-30
ErrorFcn property 20-39
ESER 4-13
even parity 6-12
event reporting 4-10
event types
  GPIB object 4-29
  serial port object 6-25
  TCP/IP object 7-21
  UDP object 7-21
events 1-7
example index 1-9
examples
communicating with a GPIB instrument 1-21
communicating with a GPIB instrument through a device object 1-25
communicating with a GPIB-VXI instrument 1-22
communicating with a serial port instrument 1-24
connecting two modems 6-31
executing a serial poll 4-39
executing a trigger, GPIB 4-36
parsing input data using scanstr 4-24
reading binary data, GPIB 4-22
reading text data versus reading binary data 3-21
recording information to disk 12-10
understanding EOI and EOS 4-25
using events and callbacks, GPIB 4-33
using events and callbacks, serial port 6-28
using software handshaking 6-35
writing and reading text data, GPIB 4-20
writing and reading text data, serial port 6-22
writing text data versus writing binary data 3-15

**F**

fclose
   disconnecting from the instrument 3-25
fclose function 18-25
fgetl
   reading text data 3-21
fgetl function 18-27
   example of 18-29
fgets
   reading text data 3-21
fgets function 18-31
   example of 18-33
FlowControl
   example of 6-35
FlowControl property 20-41
flushinput function 18-35
flushoutput function 18-36
fopen
   connecting to the instrument 3-5
fopen function 18-37
format
   record file 12-8
   serial data 6-9
fprintf
   example 4-21
   writing text data 3-16
fprintf function 18-39
fread
   example 4-23
   reading binary data 3-21
fread function 18-43
fscanf
   example 4-21
   reading text data 3-21

fscanf function 18-48
full-duplex 6-7
function handles 4-32
functions
   add 18-2
   binblockread 18-6
   binblockwrite 18-9
   clear 18-12
   clrdevice 18-13
   commit 18-14
   connect 18-15
   delete 18-17
   devicereset 18-19
   disconnect 18-20
   disp 18-22
   echotcpip 18-23
   echoudp 18-24
   fclose 18-25
   fgetl 18-27
   fgets 18-31
   flushinput 18-35
   flushoutput 18-36
   fopen 18-37
   fprintf 18-39
   fread 18-43
   fscanf 18-48
   fwrite 18-53
get 18-57
   GPIB object 3-7
geterror 18-59
gpib 18-60
   creating object 4-15
icdevice 18-64
inspect 18-69
instrcallback 18-71
   example 4-28
   example with binary data 4-34
instrfind 18-73
instrfindall 18-76
instrhelp 18-79
  introduction 1-31
instrhwinfo 18-81
  introduction 1-12
instrnotify 18-86
instrreset 18-89
invoke 18-90
isvalid 18-92
iviconfigurationstore 18-93
length 18-94
load 18-95
makemid 18-97
memmap 18-99
mempeek 18-101
mempoke 18-103
memread 18-105
memunmap 18-108
memwrite 18-110
methods 18-112
midedit 18-113
midtest 18-114
obj2mfile 18-115
propinfo 18-118
  introduction 1-31
query 18-121
readasync 18-124
  example 4-23
  example using events 4-34
record 18-127
remove 18-129
resolvehost 18-130
save 18-131
scanstr 18-133
selftest 18-136
serial 18-137
serialbreak 18-139
set 18-140
  configuring property values 3-9
  GPIB object 3-6
size 18-143
spoll 18-144
stopasync 18-146
tcip 18-148
tmtool 18-151
trigger 18-152
  example 4-37
udp 18-153
update 18-156
visa 18-159
fwrite
  example of 3-17
  writing binary data 3-16
fwrite function 18-53

G
Generic instrument drivers 11-1
  using at the command line 11-13
  using with Test & Measurement Tool 11-9
  writing 11-3
get
  GPIB object properties 3-7
get function 18-57
gterror function 18-59
gpib
  creating GPIB object 4-15
gpib function 18-60
GPIB object
  address configuration 4-18
  base properties 19-2
  callback properties 4-29
  creation 4-15
  display summary 4-16
  event types 4-29
  events and callbacks 4-28
  object-specific properties 19-5
GPIB standard 4-3
  bus and connector 4-4
  controllers 4-4
  data 4-5
data lines 4-7
enable registers 4-10
event reporting 4-10
handshake lines 4-8
interface management lines 4-7
listeners 4-4
status registers 4-10
talkers 4-4
GPIB-VXI interface 5-22
Group Execute Trigger 4-36
GUI

instredit 18-113
Property Inspector 2-6

H
half-duplex 6-7
handshake lines 4-8
HandshakeStatus
  GPIB handshake lines 4-10
HandshakeStatus property 20-43
handshaking
  hardware 6-34
  serial port object 6-33
  software 6-34
hardware handshaking 6-34
hardware resources 1-12
HardwareAssets property 20-44
help 1-31
  overview 1-9
hex2dec 12-9
hexadecimal values
  converting to decimal values 12-9
  saved to record file 12-12
high-level memory functions, VXI 5-16
HP-IB 4-3
HwIndex property 20-45
HwName property 20-46

I
icdevice function 18-64
IEEE
  488 standard 4-3
  754 standard 12-8
  format saved to record file 12-8
IFC line 4-7
inactive state, serial port 6-6
input buffer 3-20
InputBufferSize property 20-47
inspect function 18-69
instrcallback
  example 4-28
  example using binary data 4-34
instrcallback function 18-71
instrfind function 18-73
instrfindall function 18-76
instrhelp
  example of 1-31
instrhelp function 18-79
instrhmainfo
  adaptors, finding 1-13
  example of 1-12
  object constructors, finding 1-13
instrhmainfo function 18-81
instrnotify function 18-86
instrreset function 18-89
instrument control session 2-1
  loading 12-2
  saving 12-2
Instrument Control Toolbox block library
  using 16-1
Instrument Control Toolbox Block Library
  opening 16-3
instrument object 2-2
array
  creating 3-3
  example of 4-40
  configuring property values 3-9
  during object creation 3-3
connecting to instrument 2-4
creating 2-2
disconnecting from instrument 2-8
input buffer 3-20
interface-specific properties 19-5
invalid 3-25
loading 12-2
output buffer 3-14
reading data 3-19
returning from memory 18-73
returning property values 2-5
saving 12-2
specifying property names 3-10
writing data 3-13
InstrumentModel property 20-48
interface
driver adaptor 1-8
GPIB object 4-15
serial port object 6-16
TCP/IP object 7-4
UDP object 7-10
VISA-GPIB object 5-5
VISA-GPIB-VXI object 5-23
VISA-RSIB object 5-39
VISA-serial object 5-27
VISA-TCPIP object 5-35
VISA-USB object 5-31
VISA-VXI object 5-10
interface management lines 4-7
interface object 3-1
base properties 19-2
compared to device object 2-2
connecting to instrument 3-5
creating 3-2
disconnecting from instrument 3-25
returning property values 3-6
Interface property 20-49
interface-specific properties
list by object type 19-5
InterfaceIndex property 20-50
InterruptFcn property 20-51
invalid instrument object 3-25
invoke function 18-90
isvalid function 18-92
IVI drivers 10-1
constructing device object 10-9
creating MATLAB IVI driver 10-6
IVI configuration store 10-11
shared components 10-3
iviconfigurationstore function 18-93
L
LANName property 20-52
length function 18-94
listeners 4-4
load 12-4
load function 18-95
loading instrument objects
from M-file 12-3
from MAT-file 12-4
LocalHost property 20-53
LocalPort property 20-54
LocalPortMode property 20-55
logical unit 4-18
LogicalAddress property 20-56
LogicalName property 20-57
LogicalNames property 20-58
low-level memory functions, VXI 5-19
M
makemid function 18-97
ManufacturerID property 20-59
MappedMemoryBase property 20-60
MappedMemorySize property 20-62
mark parity 6-12
MasterLocation property 20-63
MAT-file
instrument objects, saving to 12-4
properties, saving to 12-2
memmap function 18-99
memory mapping, VXI 5-19
MemoryBase property 20-64
MemoryIncrement property 20-66
MemorySize property 20-69
MemorySpace property 20-71
mempeek function 18-101
mempoke function 18-103
memread function 18-105
munmap function 18-108
memwrite function 18-110
message-based communication, VXI 5-12
methods function 18-112
midedit function 18-113
midtest function 18-114
ModelCode property 20-73

N
Nagle's algorithm 20-126
Name property 20-74
    iviconfigurationstore 20-76
National Instruments
    adaptors 1-8
NDAC line 4-8
NRFD line 4-8
null modem cable 6-4

O
obj2mfile
    example 12-2
obj2mfile function 18-115
object constructor 3-2
    finding with instrhwinfo 1-13
object-specific properties 2-5
ObjectVisibility property 20-77
odd parity 6-12
online help 1-31
output buffer 3-14
output-empty event 4-30
OutputBufferSize property 20-79
OutputEmptyFcn property 20-80

P
Parent property 20-82
parity bit 6-12
Parity property 20-83
parsing input data 4-24
pin-status event 6-26
PinStatus
    example of 6-31
PinStatus property 20-85
PinStatusFcn property 20-87
Port property 20-89
PrimaryAddress
    GPIB object 4-18
    VISA-GPIB object 5-8
    VISA-GPIB-VXI object 5-26
PrimaryAddress property 20-91
ProcessLocation property 20-93
properties
    ActualLocation 20-2
    Alias 20-3
    BaudRate 20-4
    BoardIndex 20-6
    BreakInterruptFcn 20-8
    BusManagementStatus 20-10
    ByteOrder 20-12
    BytesAvailable 20-13
    BytesAvailableFcn 20-14
    BytesAvailableFcnCount 20-17
    BytesAvailableFcnMode 20-18
    BytesToOutput 20-20
    characteristics 1-31
    ChassisIndex 20-21
    CompareBits 20-23
    ConfirmationFcn 20-24
DataBits 20-25
DatagramAddress 20-26
DatagramPort 20-27
DatagramReceivedFcn 20-28
DatagramTerminateMode 20-29
DataTerminalReady 20-30
DriverName 20-31
DriverSessions 20-32
DriverType 20-33
EOIMode 20-34
EOSCharCode 20-35
EOSMode 20-36
ErrorFcn 20-39
FlowControl 20-41
HandshakeStatus 20-43
HardwareAssets 20-44
HwIndex 20-45
HwName 20-46
InputBufferSize 20-47
InstrumentModel 20-48
Interface 20-49
InterfaceIndex 20-50
InterruptFcn 20-50
LANName 20-52
LocalHost 20-53
LocalPort 20-54
LocalPortMode 20-55
LogicalAddress 20-56
LogicalName 20-57
LogicalNames 20-58
ManufacturerID 20-59
MappedMemoryBase 20-60
MappedMemorySize 20-62
MasterLocation 20-63
MemoryBase 20-64
MemoryIncrement 20-66
MemorySize 20-69
MemorySpace 20-71
ModelCode 20-73
Name 20-74
    iviconfigurationstore 20-76
ObjectVisibility 20-77
OutputBufferSize 20-79
OutputEmptyFcn 20-80
Parent 20-82
Parity 20-83
PinStatus 20-85
PinStatusFcn 20-87
Port 20-89
PrimaryAddress 20-91
ProcessLocation 20-93
PublishedAPIs 20-94
ReadAsyncMode 20-95
RecordDetail 20-97
RecordMode 20-98
RecordName 20-100
RecordStatus 20-101
RemoteHost 20-102
RemotePort 20-103
RequestToSend 20-104
Revision 20-105
RsrcName 20-106
SecondaryAddress 20-108
SerialNumber 20-110
ServerDescription 20-111
Sessions 20-112
Slot 20-113
SoftwareModules 20-114
SpecificationVersion 20-115
Status 20-116
StopBits 20-117
Tag 20-119
Terminator 20-120
Timeout 20-122
TimerFcn property 20-123
TimerPeriod property 20-125
TransferDelay 20-126
TransferStatus 20-127
TriggerFcn 20-129
Index-10

TriggerLine 20-130
TriggerType 20-131
Type 20-132
UserData 20-134
ValuesReceived 20-135
ValuesSent 20-137
Vendor 20-139
Property Inspector 2-6
   interface objects 3-11
property values
   base 2-5
      for interface objects 19-2
   configuring 3-9
      during object creation 3-3
   default 3-10
   interface-specific 19-5
   object-specific 2-5
   returning 2-5
   returning from interface objects 3-6
   saving 12-2
   specifying names 3-10
propinfo
   example of 1-31
propinfo function 18-118
PublishedAPIs property 20-94

Q
query function 18-121
Query Instrument block 21-2

R
read operations
   asynchronous 4-23
      using events 4-34
   asynchronous versus synchronous 3-23
   binary 4-22
      versus text 3-21
   completing
      GPIB object 4-20
      serial port object 6-22
      UDP object 7-15
      GPIB registers 4-14
      register-based, VXI 5-12
   synchronous 3-23
      text 4-20
      versus binary 3-21
readasync
   asynchronous read operations 3-23
   example 4-23
   example using events 4-34
readasync function 18-124
ReadAsyncMode
   asynchronous read operations 6-20
   example of 6-20
ReadAsyncMode property 20-95
record
   example 12-11
record file
   creating multiple files 12-7
   file name 12-8
   format 12-8
record function 18-127
RecordDetail
   example 12-11
   format, record file 12-8
RecordDetail property 20-97
RecordMode
   example 12-11
   multiple record files, creating 12-7
RecordMode property 20-98
RecordName
   example 12-11
   specifying a record file name 12-8
RecordName property 20-100
RecordStatus property 20-101
register-based communication, VXI 5-12
high-level memory functions, example of 5-16
low-level memory functions, example of 5-19
registers
Agilent E1432A, example of 5-17
reading and writing 4-14
serial poll 4-40
Service Request Enable 4-12
Standard Event Status 4-13
Standard Event Status Enable 4-13
Status Byte 4-12
RemoteHost property 20-102
RemotePort property 20-103
remove function 18-129
REN line 4-7
RequestToSend 20-104
resolvehost function 18-130
resource name
visa input argument 18-159
returning objects from memory 18-73
returning property values 2-5
interface object 3-6
Revision property 20-105
RI pin 6-8
RS-232 standard 6-3
RsrcName property 20-106
RTS pin 6-8
S
save function 18-131
saving instrument objects
to M-file 12-2
to MAT-file 12-4
SBR 4-12
scanstr 4-24
scanstr function 18-133
SCPI 4-3
Secondary VISA 5-4
SecondaryAddress
GPIB object 4-18
VISA-GPIB object 5-8
VISA-GPIB-VXI object 5-26
SecondaryAddress property 20-108
selftest function 18-136
serial
creating a serial port object 6-16
serial function 18-137
serial poll 4-39
serial port
configuring via operating system 6-13
connecting two devices 6-3
data format 6-9
RS-232 standard 6-3
signal and pin assignments 6-5
serial port object
base properties 19-2
callback properties 6-25
configuring communications 6-19
goald pins 6-7
using 6-30
creation 6-16
display summary 6-17
event types 6-25
events and callbacks 6-25
handshaking 6-33
object-specific properties 19-6
writing data 6-20
serialbreak function 18-139
SerialNumber property 20-110
ServerDescription property 20-111
Service Request Enable Register 4-12
SESR 4-13
session 2-1
loading 12-2
saving 12-2
Sessions property 20-112
set
configuring property values 3-9
GPIB object properties 3-6
saving property values to an M-file 12-2
set function 18-140
setserial 6-14
signal state
  GPIB 4-6
  serial port 6-6
Simulink block
  TCP/IP Receive 21-17
  TCP/IP Send 21-23
  UDP Receive 21-28
  UDP Send 21-34
Simulink Library Browser 16-4
size function 18-143
Slot property 20-113
software handshaking 6-34
SoftwareModules property 20-114
space parity 6-12
SpecificationVersion property 20-115
spoll 4-40
spoll function 18-144
SRER 4-12
SRQ line 4-7
  serial poll 4-39
Standard Event Status Enable Register 4-13
Standard Event Status Register 4-13
start bit 6-11
Status Byte Register 4-12
Status property 20-116
status registers 4-10
stop bit 6-11
stopasync function 18-146
StopBits property 20-117
stty 6-15
synchronous
  read operations 3-23
  serial port 6-10
  write operations 3-17

T
Tag property 20-119
talkers 4-4
TCP/IP object
  callback properties 7-21
  creation 7-4
  display summary 7-5
  event types 7-21
  writing data 7-14
TCP/IP Receive block 21-17
TCP/IP Send block 21-23
tcip
  creating a TCP/IP object 7-4
tcip function 18-148
TCP/IP object
  object-specific properties 19-7
termination
  EOSCharCode, example of 4-26
  read operations
    GPIB object 4-20
    serial port object 6-22
    UDP object 7-15
  Terminator, example 6-31
  write operations
    GPIB object 4-19
    serial port object 6-21
    UDP and TCIP objects 7-14
Terminator
  example 6-31
Terminator property 20-120
Test & Measurement Tool 13-1
  connecting to instrument 13-8
  exporting data 13-10
  exporting objects 13-10
  getting started 1-16
  hardware 13-4
  instrument drivers 13-16
  instrument objects 13-12
  saving the session 13-11
text
Index

read operations 4-20
versus binary 3-21
write operations 4-20
versus binary 3-15
Timeout property 20-122
timer event 4-30
tmtool
  See also Test & Measurement Tool 13-1
tmtool function 18-151
To Instrument block 21-10
toolbox components
  interface driver adaptor 1-8
  M-files 1-7
TransferDelay property 20-126
TransferStatus property 20-127
trigger
  example of 4-37
trigger function 18-152
TriggerFcn property 20-129
TriggerLine property 20-130
TriggerType property 20-130
troubleshooting
  serial ports 6-13
tutorials 1-9
Type property 20-132

U
udp
  creating a UDP object 7-10
udp function 18-153
UDP object
  callback properties 7-21
  creation 7-10
  display summary 7-11
  event types 7-21
  object-specific properties 19-7
  writing data 7-14
UDP Receive block 21-28
UDP Send block 21-34
update function 18-156
UserData
  saving values to a MAT-file 12-2
UserData property 20-134

V
values versus bytes 3-14
ValuesReceived property 20-135
ValuesSent property 20-137
Vendor property 20-139
VISA
  secondary 5-4
visa function 18-159
VISA-GPIB object
  address configuration 5-7
  base properties 19-2
  creation 5-5
  display summary 5-7
  object-specific properties 19-8
VISA-GPIB-VXI object
  address configuration 5-25
  base properties 19-2
  creation 5-23
  display summary 5-24
  object-specific properties 19-8
VISA-RSIB object
  address configuration 5-41
  creation 5-39
  display summary 5-40
  object-specific properties 19-9
VISA-serial object
  base properties 19-2
  communication configuration 5-29
  creation 5-27
  display summary 5-28
  object-specific properties 19-10
VISA-TCPIP object
  address configuration 5-37
  creation 5-35

display summary 5-36
object-specific properties 19-10

VISA-USB object
address configuration 5-33
creation 5-31
display summary 5-32
object-specific properties 19-11

VISA-VXI object
address configuration 5-12
base properties 19-2
creation 5-10
display summary 5-11
object-specific properties 19-11
register-based communication 5-12

VXI interface 5-9
VXI\textit{plug\&play} drivers 9-1
constructing device object 9-8
creating MATLAB VXI\textit{plug\&play} driver 9-5

Instrument Help 1-31
Property Inspector 2-6
write operations
asynchronous 3-17
binary 3-15
completing
GPIB object 4-19
serial port object 6-21
UDP and TCP/IP objects 7-14

GPIB registers
common commands 4-14
serial poll example 4-40

register-based, VXI 5-12
synchronous 3-17
text
element 4-20
text versus binary 3-15
values versus bytes 3-14

Workspace browser
Display Hardware Info 1-15
Display Summary 4-16

Xoff 6-34
Xon 6-34