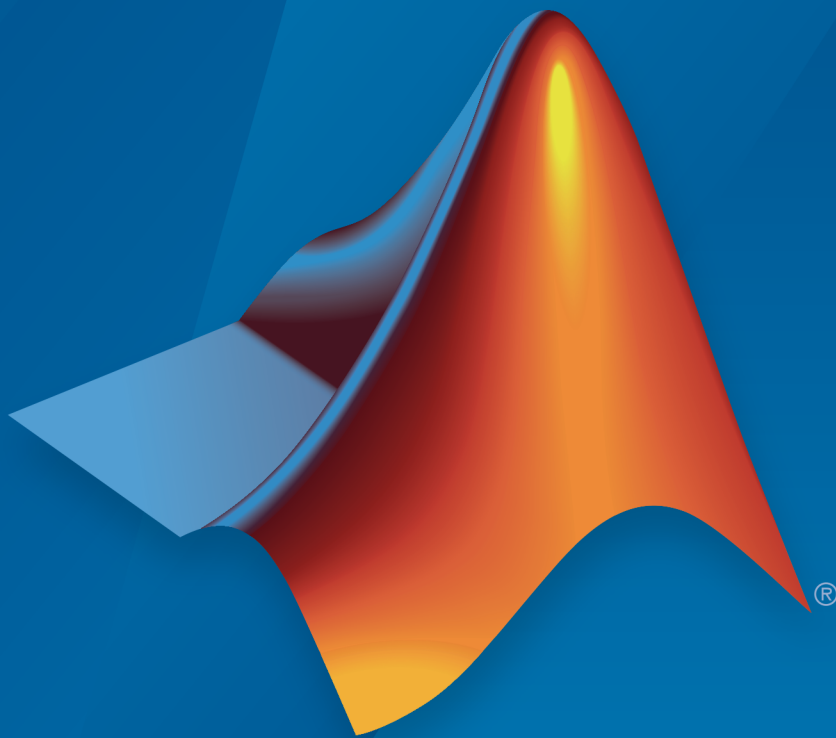# Instrument Control Toolbox™

User's Guide

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

Contents

Instrument Control Session

2

Using Interface Objects

3

4

Controlling Instruments Using VISA

5

Controlling Instruments Using the Serial Port

6

Controlling Instruments Using TCP/IP and UDP

7

Controlling Instruments Using Bluetooth

8

Controlling Instruments Using I2C

9

Controlling Instruments Using SPI

10

13

Instrument Support Packages

14

Using Generic Instrument Drivers

15

Saving and Loading the Session

16

Using the Instrument Driver Testing Tool

19

Instrument Control Toolbox Troubleshooting

20

Using the Instrument Control Toolbox Block Library

21

22 Functions — Alphabetical List

23 Properties — Alphabetical List

24 Block Reference

A Vendor Driver Requirements and Limitations

B

Getting Started

Instrument Control Toolbox Product Description

Control and communicate with test and measurement instruments

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

With Instrument Control Toolbox, you can generate data in MATLAB to send out to an instrument, or read data into MATLAB for analysis and visualization. You can automate tests, verify hardware designs, and build test systems based on LXI, PXI, and AXIe standards. For remote communication with other computers and devices from MATLAB, the toolbox provides built-in support for TCP/IP, UDP, I2C and Bluetooth® serial protocols.

Key Features

- IVI, VXI*plug&play*, and native MATLAB instrument driver support
- GPIB and VISA (GPIB, GPIB-VXI, VXI, USB, TCP/IP, and serial) support
- TCP/IP, UDP, I2C and Bluetooth serial protocol support
- Instrument Control app for identifying, configuring, and communicating with instruments
- Simulink® blocks for sending and receiving live data between instruments and Simulink models
- Functions for reading and writing binary and ASCII data to and from instruments
- Synchronous and asynchronous (blocking and nonblocking) read-and-write operations

Instrument Control Toolbox Overview

In this section...
"Getting to Know the Instrument Control Toolbox Software" on page 1-3
"Exploring the Instrument Control Toolbox Software" on page 1-4
"Learning About the Instrument Control Toolbox Software" on page 1-4
"Using the Documentation Examples" on page 1-5

Getting to Know the Instrument Control Toolbox Software

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

- A framework for communicating with instruments that support the GPIB interface (IEEE®-488), the VISA standard, and the TCP/IP and UDP protocols. Note that the toolbox extends the basic serial port features included with the MATLAB software.

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

- Functions for transferring data between the MATLAB workspace and your instrument:

  - The data can be binary (numerical) or text.

  - The transfer can be synchronous and block access to the MATLAB Command Window, or asynchronous and allow access to the MATLAB Command Window.

- Event-based communication.

- Functions for recording data and event information to a text file.

- Tools that facilitate instrument control in an easy-to-use graphical environment.

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

http://www.mathworks.com/agilentcmdexpert

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

http://www.mathworks.com/products/instrument/related.html.

Exploring the Instrument Control Toolbox Software

For a list of the toolbox functions, type

```
help instrument
```

For the code of a function, type

```
type function_name
```

For help for any function, type

```
instrhelp function_name
```

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

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

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

Learning About the Instrument Control Toolbox Software

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

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

Using the Documentation Examples

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

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

About Instrument Control

In this section...
"Passing Information Between the MATLAB Workspace and Your Instrument" on page 1-6
"MATLAB Functions" on page 1-8
"Interface Driver Adaptor" on page 1-8

Passing Information Between the MATLAB Workspace and Your Instrument

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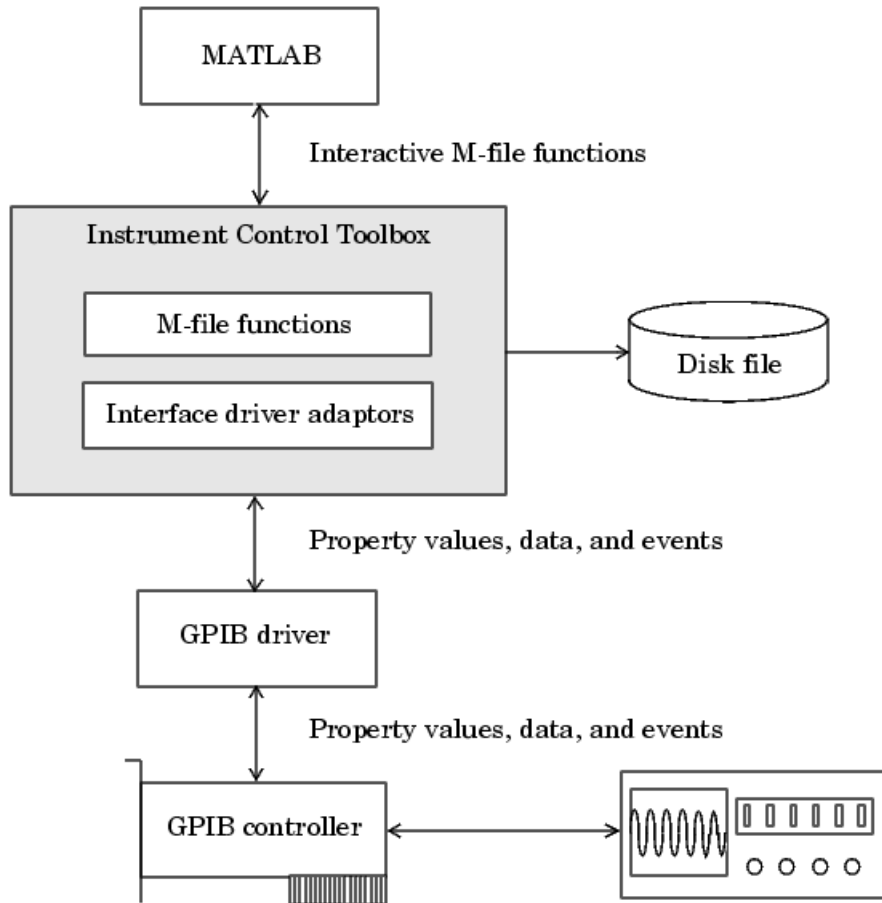
This diagram illustrates how information flows from component to component. Information consists of

- **Property values**

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

- **Data**

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

- **Events**

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

MATLAB Functions

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

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

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

```
help instrument
```

Interface Driver Adaptor

The interface driver adaptor (or just *adaptor*) is the link between the toolbox and the interface driver. The adaptor's main purpose is to pass information between the MATLAB workspace and the interface driver. Interface drivers are provided by your instrument vendor. For example, if you are communicating with an instrument using a

National Instruments GPIB controller, then an interface driver such as NI-488.2 must be installed on your platform. Note that interface drivers are not installed as part of the Instrument Control Toolbox software.

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

Interface Adaptors

Interface	Adaptor Name
GPIB	`advantech`, `agilent`, `cec`, `contec`, `ics`, `iotech`, `keithley`, `mcc`, `ni`
Serial port	N/A
TCP/IP	N/A
UDP	N/A
VISA standard	`agilent`, `ni`, `tek`

As described in "Examining Your Hardware Resources" on page 1-14, you can list the supported interfaces and adaptor names with the `instrhwinfo` function.

Installation Information

In this section...
"Installation Requirements" on page 1-10
"Toolbox Installation" on page 1-10
"Hardware and Driver Installation" on page 1-10

Installation Requirements

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

- MATLAB
- Instrument Control Toolbox software

Additionally, you might need to install hardware such as a GPIB controller and vendor-specific software such as drivers, support libraries, and so on. For a complete list of all supported vendors, refer to "Interface Driver Adaptor" on page 1-8.

Toolbox Installation

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

```
ver
```

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

For information about installing the toolbox, refer to the installation documentation for your platform. If you experience installation difficulties, look for the installation and license information at the MathWorks® Web site (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.

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

Supported Hardware

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

Feature	64-bit MATLAB on Windows	32-bit MATLAB on Windows	64-bit MATLAB on Mac OS	64-bit MATLAB on Linux
Serial	supported	supported	supported	supported
TCP/IP	supported	supported	supported	supported
UDP	supported	supported	supported	supported
VISA [3]	supported [1]	supported [1]		supported [1]
GPIB [4]	supported [1]	supported [1]		supported [1]
I2C [5]	supported [1]	supported [1]	supported [1]	supported [1]
SPI [5]	supported [1]	supported [1]	supported [1]	supported [1]
Bluetooth [6]	supported	supported	supported [7]	
Quick-Control Oscilloscope and Quick-Control Function Generator	supported [2]	supported [2]	supported [2]	supported [2]
MATLAB Instrument Drivers	supported	supported	supported	supported
MATLAB Instrument Drivers made using IVI-C drivers and Instrument Wrappers for IVI-C drivers	supported [1]	supported [1]		
MATLAB Instrument Drivers made		supported [1]		

Feature	64-bit MATLAB on Windows	32-bit MATLAB on Windows	64-bit MATLAB on Mac OS	64-bit MATLAB on Linux
using IVI-COM drivers and Instrument Wrappers for IVI-COM drivers				

Table Notes

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

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

3. Requires Agilent, National Instruments, Tektronix, or TAMS VISA compliant with VISA specification 5.0 or higher.

4. Requires Advantech®, Agilent, CEC, Contec, ICS Electronics™, IOtech®, Keithley®, MCC, or National Instruments hardware and driver.

5. Requires Aardvark or National Instruments hardware and driver.

6. Bluetooth Serial Port Profile only.

7. Bluetooth on 64-bit MATLAB on Mac OS 10.7 and earlier versions only.

Examining Your Hardware Resources

In this section...
"instrhwinfo Function" on page 1-14
"Test & Measurement Tool" on page 1-18
"Viewing the IVI Configuration Store" on page 1-19

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 1-14
- "Interface Information" on page 1-14
- "Adaptor Information" on page 1-15
- "Instrument Object Information" on page 1-16
- "Installed Driver Information" on page 1-17

General Toolbox Information

For general information about the Instrument Control Toolbox, type:

```
instrhwinfo

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

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

Interface Information

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

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

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

    InstalledAdaptors: {'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`.

Interface Name	Adaptor Name
gpib	advantech, agilent, cec, contec, ics, iotech, keithley, mcc, ni
visa	agilent, ni, tek

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

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

ghwinfo =

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

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

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

To display information for the Tektronix VISA adaptor,

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

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

The available VISA object constructor names are shown below.

```
vhwinfo.ObjectConstructorName
ans =

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

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

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

Instrument Object Information

To display information about a specific instrument object, you supply the object as an argument to `instrhwinfo`. For example, to display information for the GPIB object created in the ("Adaptor Information" on page 1-15), type:

```
ghwinfo = instrhwinfo(g)
ghwinfo =

            AdaptorDllName: [1x82 char]
         AdaptorDllVersion: 'Version 2.0 (R14)'
```

```
              AdaptorName: 'NI'
            VendorDllName: 'gpib-32.dll'
    VendorDriverDescription: 'NI-488'
```

To display information for the VISA-GPIB object created in the ("Adaptor Information" on page 1-15), type:

```
vghwinfo = instrhwinfo(vg)
vghwinfo =

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

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

Installed Driver Information

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

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

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

```
instrhwinfo('vxipnp', 'ag3325b')
ans =
      Manufacturer: 'Agilent Technologies'
             Model: 'Agilent 3325B Synthesizer/Func. Gen.'
     DriverVersion: '4.1'
     DriverDllName: 'C:\VXIPNP\WINNT\bin\ag3325b_32.dll'
```

Test & Measurement Tool

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

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

To open the Test & Measurement Tool, type:

```
tmtool
```

Hardware

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

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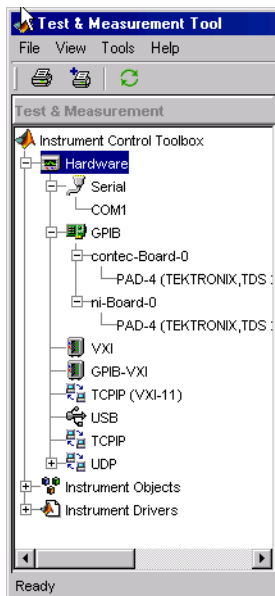
Installed Drivers

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

Right-click the Instrument Drivers node to scan for installed drivers. The driver-type nodes expand to include entries for each driver found by the scan. Note that for MATLAB instrument drivers and VXI*plug&play* drivers, the installation of a driver requires only the presence of a driver file. For IVI, installation involves an IVI configuration store; see "Viewing the IVI Configuration Store" on page 1-19.
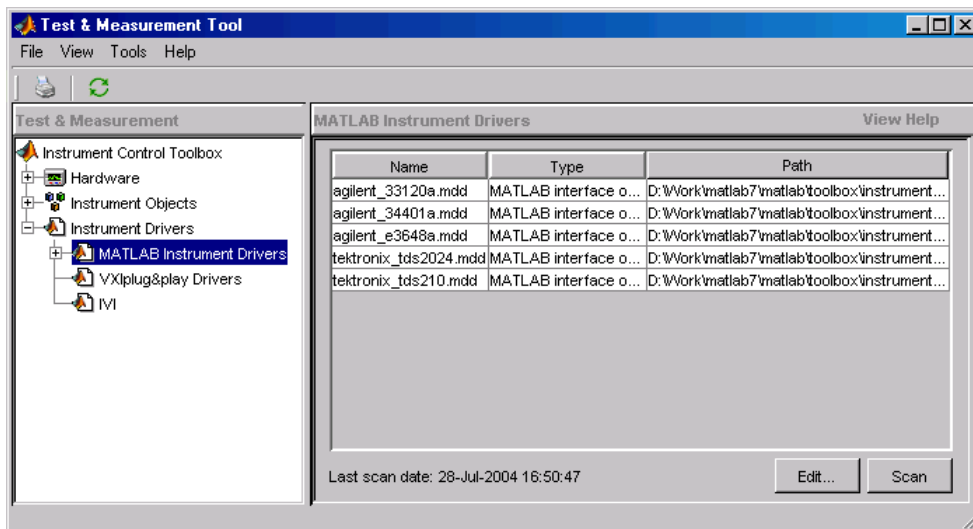
The Test & Measurement Tool GUI includes embedded help. For further details about the Test & Measurement Tool and its capabilities, see "Test & Measurement Tool Overview" on page 17-2.

Viewing the IVI Configuration Store

An IVI configuration store greatly enhances instrument interchangeability by providing the means to configure the relationship between drivers and I/O interface references outside of the application. For details of the components of an IVI configuration store, see "IVI Configuration Store" on page 13-16.

Command-Line Configuration

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

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

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

```
instrhwinfo('ivi','MainScope')
ans =
                DriverSession: 'TekScope.DriverSession'
                HardwareAsset: 'TekScope.Hardware'
               SoftwareModule: 'TekScope.Software'
        IOResourceDescriptor: 'GPIB0::13::INSTR'
SupportedInstrumentModels: 'TekScope 5000, 6000 and 7000 series'
           ModuleDescription: 'TekScope software module desc'
              ModuleLocation: ''
```

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

Using the Test & Measurement Tool

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

```
tmtool
```

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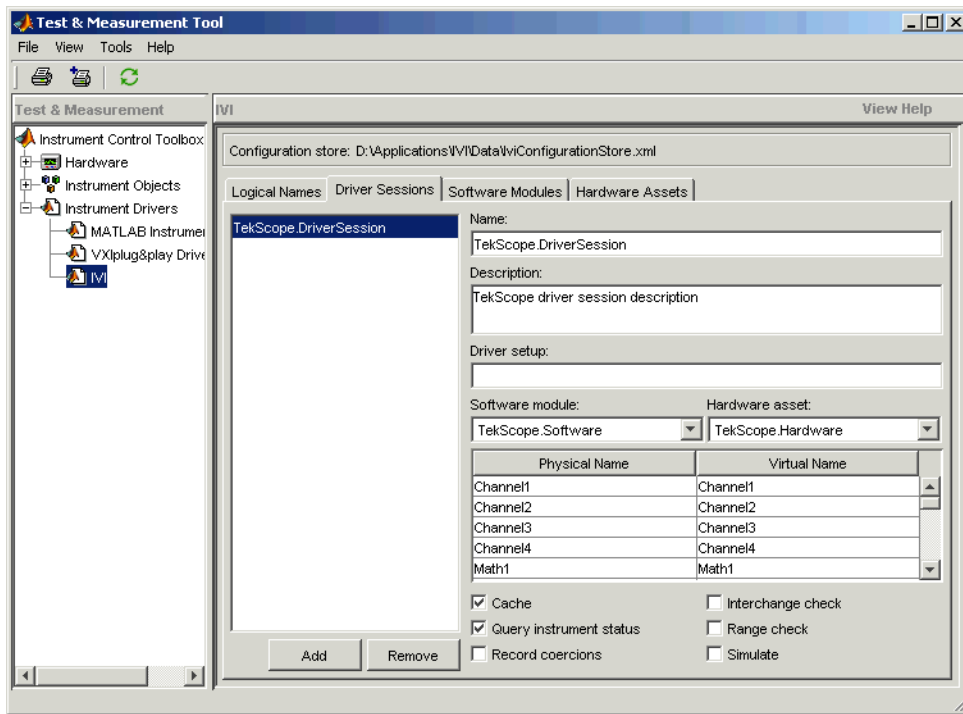
You see a tab for each type of IVI configuration store element. This figure shows the available driver sessions in the current IVI configuration store. For the selected driver session, you can use any available software module or hardware asset. This figure shows the configuration for the driver session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

Communicating with Your Instrument

In this section...

Instrument Control Session Examples

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

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

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

Communicating with a GPIB Instrument

This example illustrates how to communicate with a GPIB instrument. The GPIB controller is a National Instruments AT-GPIB card. The instrument is 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 "GPIB Overview" on page 4-2.

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

```
g = gpib('ni',0,1);
```
2 **Connect to the instrument** — Connect `g` to the instrument.

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

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

```
fprintf(g,'Volt 3')
fprintf(g,'Volt?')
data = fscanf(g)
data =
+3.00000E+00
```
5 **Disconnect and clean up** — When you no longer need `g`, you should disconnect it from the instrument, remove it from memory, and remove it from the MATLAB workspace.

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

Communicating with a GPIB-VXI Instrument

This example illustrates how to communicate with a VXI instrument via a GPIB controller using the VISA standard provided by 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 "VISA Overview" on page 5-2.

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

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

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

```
fopen(v)
```

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

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

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

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

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

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

Communicating with a Serial Port Instrument

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

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

Note: This example is Windows specific.

---

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

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

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

```
s.BaudRate = 4800
s.Terminator = 'CR'
```

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

```
fopen(s)
```

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

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

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

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

Communicating with a GPIB Instrument Using a Device Object

This example illustrates how to communicate with a GPIB instrument through a device object. The GPIB controller is a 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 "Device Objects" on page 11-2.

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`.

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

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

```
connect(d)
```

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

```
devicereset(d)
```

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

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

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

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

General Preferences for Instrument Control

In this section...
"Accessing General Preferences" on page 1-27
"MATLAB Instrument Driver Editor" on page 1-28
"MATLAB Instrument Driver Testing Tool" on page 1-29
"Device Objects" on page 1-30
"IVI Configuration Store" on page 1-31
"IVI Instruments" on page 1-31

Accessing General Preferences

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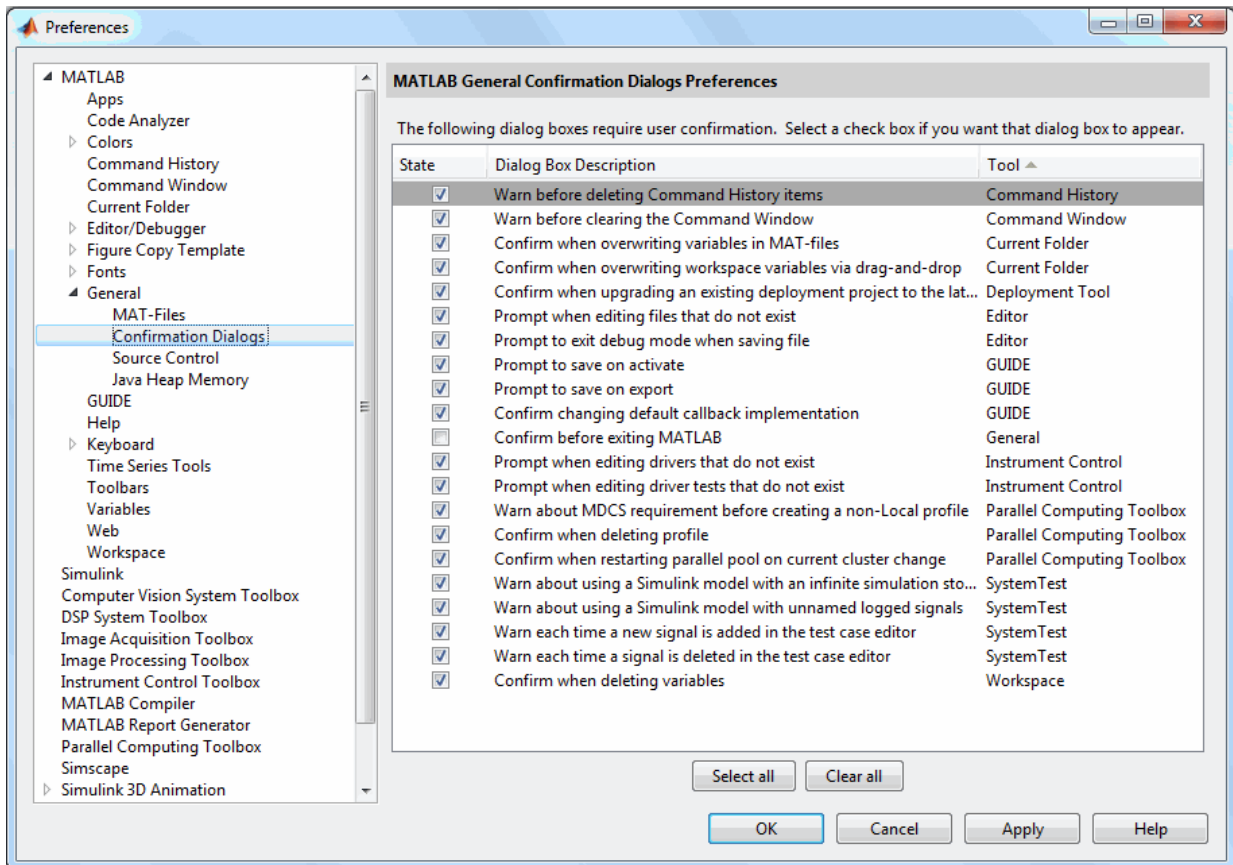
MATLAB Instrument Driver Editor

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

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

For example, the command

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

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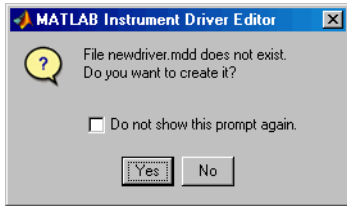
If you select **Do not show this prompt again**, the corresponding check box in the Preferences dialog box is cleared, in which case the MATLAB Instrument Driver Editor creates new driver files without prompting. To reactivate the prompt, select the option on the Preferences dialog box.

MATLAB Instrument Driver Testing Tool

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

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

For example, the command

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

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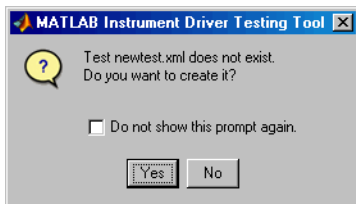
If you select **Do not show this prompt again**, the corresponding check box in the Preferences dialog box is cleared, in which case the MATLAB Instrument Driver Testing Tool creates new driver test files without prompting. To reactivate the prompt, check the option on the Preferences dialog box.

Device Objects

You access other Instrument Control Preferences by selecting the **Instrument Control Toolbox** node in the tree.
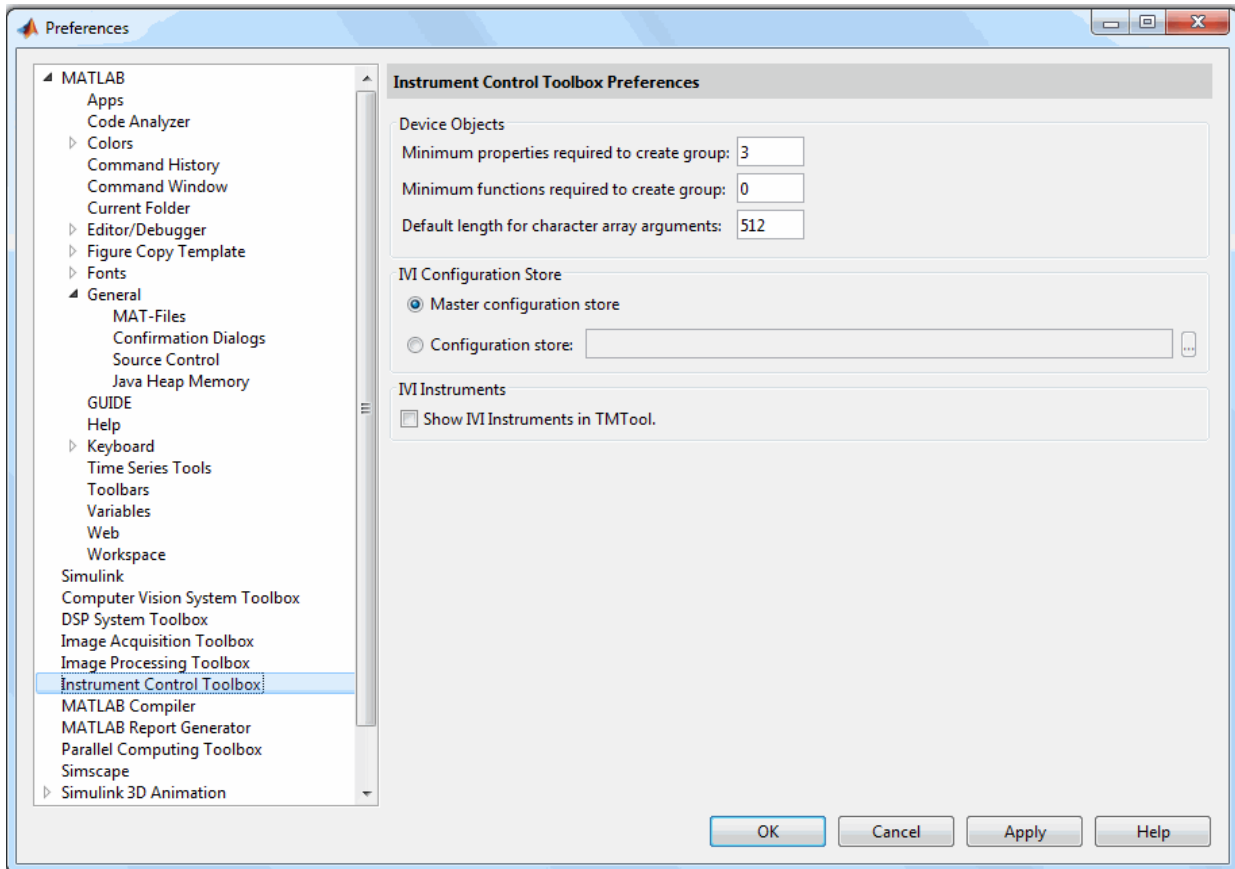
The **Device Objects** section of the dialog box contains preferences related to the construction and use of device objects for VXI*plug&play* and IVI-C drivers.

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

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

IVI Configuration Store

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

IVI Instruments

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

For more information, see "IVI-C Class Compliant Wrappers in Test & Measurement Tool" on page 13-24.

Interface and Property Help

In this section...

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

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

propinfo Function

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

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

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

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

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

This information tells you the following:

- The property value data type is a 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,

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

instrsupport Function

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

Overview Help

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

```
help instrument
```

For the code for any function, type

```
type function_name
```

Documentation Examples

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

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

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

Online Support

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

2

Instrument Control Session

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

Creating Instrument Objects

Overview

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

- Interface objects — 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 "Creating an Interface Object" on page 3-2.

Device Objects

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

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

For detailed information on device objects and how to create and use them, see "Device Objects" on page 11-2.

Connecting to the Instrument

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

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

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

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

Configuring and Returning Properties

In this section...
"Configuring Property Names and Property Values" on page 2-5
"Returning Property Names and Property Values" on page 2-5
"Property Inspector" on page 2-5

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 "MATLAB Instrument Driver Editor Overview" on page 18-2.

Disconnecting and Cleaning Up

In this section...
"Disconnecting an Instrument Object" on page 2-8
"Cleaning Up the MATLAB Workspace" on page 2-8

Disconnecting an Instrument Object

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

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

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

Cleaning Up the MATLAB Workspace

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

```
delete(obj)
```

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

```
clear obj
```

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

Summary

In this section...
"Advantages of Using Device Objects" on page 2-9
"When to Use Interface Objects" on page 2-9

Advantages of Using Device Objects

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

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

- You do not need to know instrument-specific commands
- You can use standard VXI*plug&play* or IVI instrument drivers provided by your instrument vendor or other party
- You can use a MATLAB instrument driver to control your instrument. To get a MATLAB instrument driver, you can

  - Convert a VXI*plug&play* or IVI driver
  - Use a MATLAB driver that is shipped with the toolbox
  - Create it yourself or modify a similar driver
  - Install it from a third party, such as MATLAB Central

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

When to Use Interface Objects

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

- Your instrument does not have a standard instrument driver supported by the Instrument Control Toolbox software.

- You are using a streaming application (typically serial, UDP, or TCP/IP interface) to notify you of some occurrence.
- Your application requires frequent changes to communication channel settings.

Instrument Control Toolbox Properties

The following properties are available in the toolbox.

- `ActualLocation`
- `Alias`
- `BaudRate`
- `BoardIndex`
- `BreakInterruptFcn`
- `BusManagementStatus`
- `ByteOrder`
- `BytesAvailableFcn`
- `BytesAvailableFcnCount`
- `BytesAvailableFcnMode`
- `BytesToOutput`
- `ChassisIndex`
- `CompareBits`
- `ConfirmationFcn`
- `DataBits`
- `DatagramAddress`
- `DatagramPort`
- `DatagramReceivedFcn`
- `DatagramTerminateMode`
- `DataTerminalReady`
- `DriverName`
- `DriverSessions`
- `DriverType`
- `EOIMode`
- `EOSCharCode`
- `EOSMode`
- `ErrorFcn`

- FlowControl
- HandshakeStatus
- HardwareAssets
- HwIndex
- HwName
- InputBufferSize
- InputDatagramPacketSize
- InstrumentModel
- Interface
- InterfaceIndex
- InterruptFcn
- LANName
- LocalHost
- LocalPort
- LocalPortMode
- LogicalAddress
- LogicalName
- LogicalNames
- ManufacturerID
- MappedMemoryBase
- MappedMemorySize
- MasterLocation
- MemoryBase
- MemoryIncrement
- MemorySize
- MemorySpace
- ModelCode
- Name
- NetworkRole
- ObjectVisibility

- OutputBufferSize
- OutputDatagramPacketSize
- OutputEmptyFcn
- Parent
- Parity
- PinStatus
- PinStatusFcn
- Port
- PrimaryAddress
- ProcessLocation
- PublishedAPIs
- ReadAsyncMode
- RecordDetail
- RecordMode
- RecordName
- RecordStatus
- RemoteHost
- RemotePort
- RequestToSend
- Revision
- RsrcName
- SecondaryAddress
- SerialNumber
- ServerDescription
- Sessions
- Slot
- SoftwareModules
- SpecificationVersion
- Status
- StopBits

- `Tag`
- `Terminator`
- `Timeout`
- `TimerFcn`
- `TimerPeriod`
- `TransferDelay`
- `TransferStatus`
- `TriggerFcn`
- `TriggerLine`
- `TriggerType`
- `Type`
- `UserData`
- `ValuesReceived`
- `ValuesSent`
- `Vendor`

Using Interface Objects

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

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 functions called *object creation functions* (or *object constructors*). These files are implemented using MATLAB object-oriented programming capabilities, which are described in the MATLAB documentation.

Interface Object Creation Functions

Constructor	Description
gpib	Create a GPIB object.
serial	Create a serial port object.
tcpip	Create a TCPIP object.
udp	Create a UDP object.
visa	Create a VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, or VISA-serial object.
bluetooth	Create a Bluetooth object.
i2c	Create an I2C object.

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,

```
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`:

```
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 the properties documentation.

Creating an Array of Instrument Objects

In the MATLAB workspace, 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`:

```
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`:

```
x = [g1 g2]
Instrument Object Array

   Index:  Type:      Status:     Name:
   1       gpib       closed      GPIB0-1
   2       gpib       closed      GPIB0-2
```

To create the column array `y`:

```
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.

```
x.EOSMode = 'read'
```

Refer to the functions documentation 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.

```
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 the properties documentation.

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.Status
ans =
open
```

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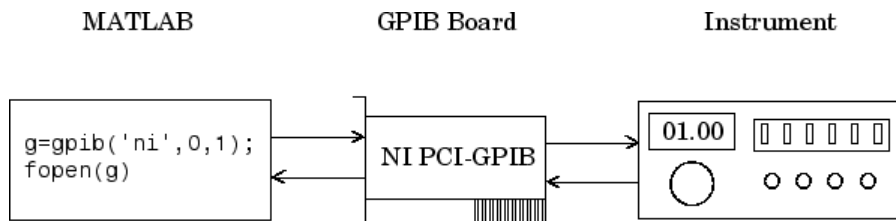
Configuring and Returning Properties

In this section...
"Base and Interface-Specific Properties" on page 3-6
"Returning Property Names and Property Values" on page 3-6
"Configuring Property Values" on page 3-9
"Specifying Property Names" on page 3-9
"Default Property Values" on page 3-9
"Property Inspector" on page 3-10

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 set configurable properties. 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.

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

```
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 = O
ValuesSent = O

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

To display the current value for one property, you supply the property name to `get`.

```
g.OutputBufferSize
ans =
   512
```

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

```
g.BoardIndex
ans =
   [0]

g.TransferStatus
ans =
   'idle'
```

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

```
g.PrimaryAddress
ans =
```

1

Configuring Property Values

You can configure property values using the object

```
g.EOSMode = 'read'
```

To configure values for multiple properties, you can set each one as follows.

```
g.EOSCharCode = 'CR'
g.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 the properties documentation 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.

```
g.EOSMode = 'read'
g.eosmode = 'read'
g.EOSM = 'read'
```

However, when you include property names in a file, you should use the full property name. This practice can prevent problems with future releases of the Instrument Control Toolbox software 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`.

```
g.EOSMode
ans =

none
```

You can also use the `propinfo` function, or refer to the functions documentation 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 3-18

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 the MATLAB workspace to the instrument, and from the instrument to the MATLAB workspace?

  The Instrument Control Toolbox automatically manages the data transferred between the MATLAB workspace and the instrument. For many common applications, you can ignore the buffering and data flow process. However, if you are transferring a large number of values, 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 Window?

  You control access to the MATLAB Command Window 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 line, 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

Function Name	Description
binblockwrite	Write binblock data to the instrument.
fprintf	Write text to the instrument.
fwrite	Write binary data to the instrument.
stopasync	Stop asynchronous read and write operations.

Properties Associated with Writing Data

Property Name	Description
BytesToOutput	Indicate the number of bytes currently in the output buffer.
OutputBufferSize	Specify the size of the output buffer in bytes.
Timeout	Specify the waiting time to complete a read or write operation.
TransferStatus	Indicate if an asynchronous read or write operation is in progress.
ValuesSent	Indicate the total number of values written to the instrument.

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 the MATLAB workspace 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:

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

```
g.OutputBufferSize

ans =

   512
g.BytesToOutput

ans =

     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.
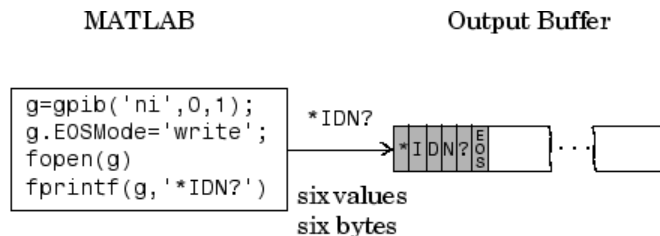
□ Bytes used during write
□ Bytes unused during write

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.

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.
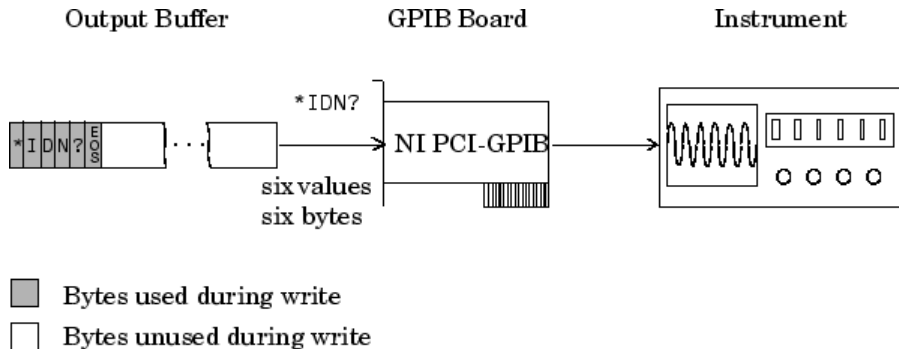
Writing Text Data Versus Writing Binary Data

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\n` 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.

```
g = gpib('ni',0,1);
g.OutputBufferSize = 3000;
```
2   **Connect to the instrument** — Connect `g` to the instrument.

```
fopen(g)
```

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

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

Create the waveform data.

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

Write the binary waveform data to the scope.

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

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

```
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.

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

Synchronous Versus Asynchronous Write Operations

By default, all write functions operate synchronously and block the MATLAB Command Window 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.

```
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.

```
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.

```
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.

```
g.BytesToOutput
ans =
        2512
```

Reading Data

Functions Associated with Reading Data

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

Properties Associated with Reading Data

Property Name	Description
BytesAvailable	Indicate the number of bytes available in the input buffer.
InputBufferSize	Specify the size of the input buffer in bytes.
ReadAsyncMode	Specify whether an asynchronous read is continuous or manual (serial port, TCPIP, UDP, and VISA-serial objects only).
Timeout	Specify the waiting time to complete a read or write operation.
TransferStatus	Indicate if an asynchronous read or write operation is in progress.
ValuesReceived	Indicate the total number of values read from the instrument.

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 the MATLAB workspace 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 a 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:

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

ans =

   512
g.BytesAvailable

ans =

    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.

Bytes used during read

Bytes unused during read

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`.

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.

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.

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

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

```
fopen(g)
```

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

```
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.

```
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 period value to a string, use the `char` function.

```
char(period)
ans =
1.006E-3
```

The `ValuesReceived` property indicates the total number of values that were read from the instrument.

```
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.

```
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 Window 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.

```
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 the MATLAB workspace.

```
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);
```

Using SCPI Commands

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

Commonly Used SCPI Commands

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

Disconnecting and Cleaning Up

In this section...
"Disconnecting an Instrument Object" on page 3-24
"Cleaning Up the MATLAB Workspace" on page 3-24

Disconnecting an Instrument Object

When you no longer need an instrument object, you should disconnect it from the instrument, and clean up the MATLAB workspace 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 Workspace

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 the MATLAB workspace with the `instrfind` function.

4

Controlling Instruments Using GPIB

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

GPIB Overview

In this section...

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 4-13. If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to "Creating Instrument Objects" on page 2-2.

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

Important GPIB Features

The important GPIB features are described below. For detailed information about GPIB functionality, see the appropriate references in the Appendix B.

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 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.
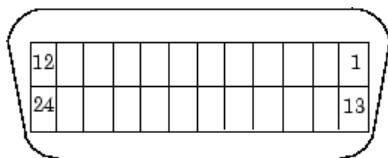
GPIB Pin and Signal Assignments

Pin	Label	Signal Name	Pin	Label	Signal Name
1	DIO1	Data transfer	13	DIO5	Data transfer

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

Data Lines

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

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

Interface Management Lines

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

GPIB Interface Management Lines

Line	Description
ATN	Used by the Controller to inform all devices on the GPIB that bytes are being sent. If the ATN line is high, the bytes are interpreted as an instrument command. If the ATN line is low, the bytes are interpreted as an interface message.
IFC	Used by the Controller to initialize the bus. If the IFC line is low, the Talker and Listeners are unaddressed, and the System Controller becomes the Controller-In-Charge.

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

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.

Handshake Lines

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

The handshaking process follows these steps:

1 Initially, the Talker holds the DAV line high indicating no data is available, while the Listeners hold the NRFD line high and the NDAC line low indicating they are ready for data and no data is accepted, respectively.

2 When the Talker puts data on the bus, it sets the DAV line low, which indicates that the data is valid.

3 The Listeners set the NRFD line low, which indicates that they are not ready to accept new data.

4 The Listeners set the NDAC line high, which indicates that the data is accepted.

5 When all Listeners indicate that they have accepted the data, the Talker sets the DAV line high indicating that the data is no longer valid. The next byte of data can now be transmitted.

6 The Listeners hold the NRFD line high indicating they are ready to receive data again, and the NDAC line is held low indicating no data is accepted.

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

The handshaking steps are shown below.
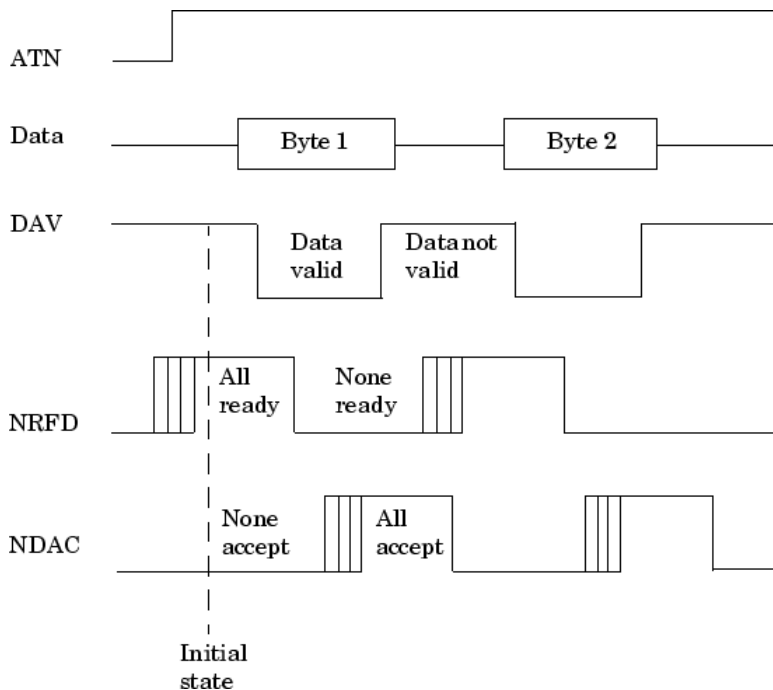
You can examine the state of the handshake lines with the `HandshakeStatus` property.

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

bit 7   bit 0

| PON | URQ | CME | EXE | DDE | QYE | RQC | OPC | Standard Event Status Register

| PON | URQ | CME | EXE | DDE | QYE | RQC | OPC | Standard Event Status Enable
                                                                    Register

Logical OR

Output Queue

Data available on instrument
ready to be sent across the bus →

bit 7   bit 0

| — | RQS / MSS | ESB | MAV | — | — | — | — | Status Byte Register

| — | ⊠ | ESB | MAV | — | — | — | — | Service Request Enable
                                                   Register

Logical OR
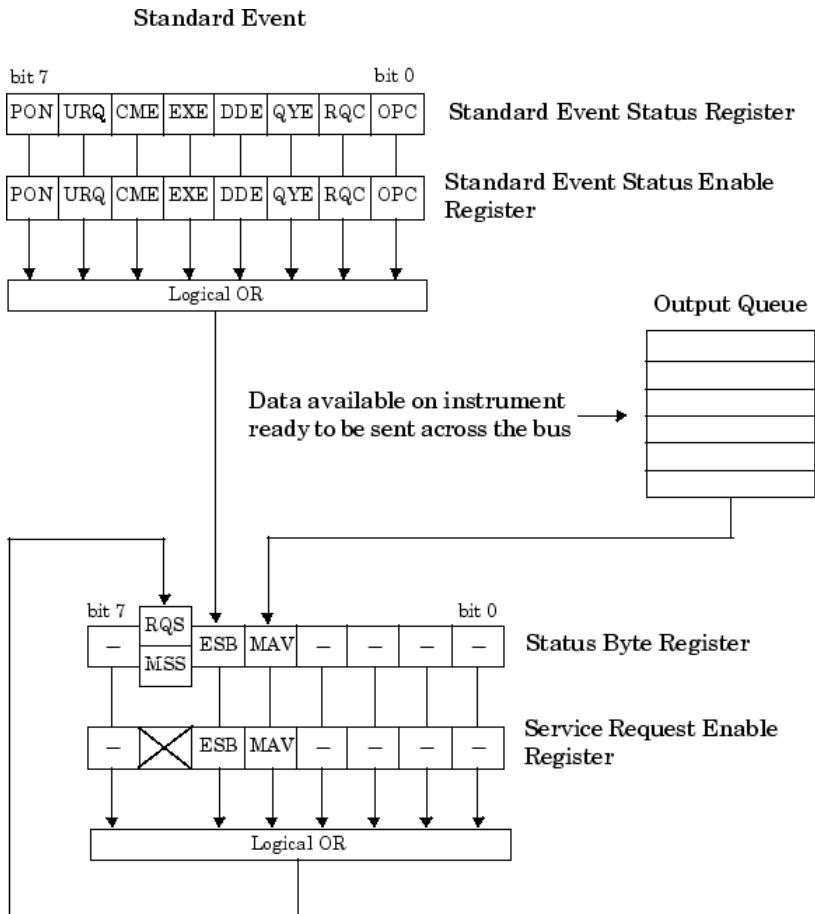
Status Byte Register

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

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

For example, if you want to know when a specific type of instrument error occurs, 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 4-11) 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.

SESR Bits

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

For example, if you want to know when an execution error occurs, you would enable bit 4 of the ESER. Additionally, you would enable bit 5 of the SRER (see "Status Byte Register" on page 4-10) so that the error event of interest is reported by the ESB bit of the SBR.

Reading and Writing Register Information

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

Register Commands

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

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

To see how to use many of these commands in the context of an instrument control session, refer to "Executing a Serial Poll" on page 4-39.

Creating a GPIB Object

In this section...
"Using the gpib Function" on page 4-13
"GPIB Object Display" on page 4-14

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,

```
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 IEEE workspace. You can display the class of g with the whos command.

```
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

Property Name	Description
`Name`	Specify a descriptive name for the GPIB object.
`Type`	Indicate the object type.

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

```
g.Name

ans =

GPIB0-1

g.Type

ans =

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:    O
    ValuesReceived:    O
    ValuesSent:        O
```

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 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

Property Name	Description
BoardIndex	Specify the index number of the GPIB board.
PrimaryAddress	Specify the primary address of the GPIB instrument.
SecondaryAddress	Specify the secondary address of the GPIB instrument.

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 4-13 .

```
g.BoardIndex

ans =

     0

g.PrimaryAddress

ans =

     1
```

```
g.SecondaryAddress

ans =

    0
```

Writing and Reading Data

In this section...

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.

Writing and Reading Text Data

These functions are used when reading and writing text:

Function	Purpose
`fprintf`	Write text to an instrument.
`fscanf`	Read data from an instrument and format as text.

These properties are associated with reading and writing text:

Property	Purpose
`ValuesReceived`	Specifies the total number of values read from the instrument.
`ValuesSent`	Specifies the total number of values sent to the instrument.
`InputBufferSize`	Specifies the total number of bytes that can be queued in the input buffer at one time.
`OutputBufferSize`	Specifies the total number of bytes that can be queued in the output buffer at one time.
`EOSMode`	Configures the End-Of-String termination mode.
`EOSCharCode`	Specifies the End-Of-String terminator.

Property	Purpose
EOIMode	Enables or disables the assertion of the EOI mode at the end of a write operation.

The following 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.

```
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.

```
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.

```
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.

```
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.

```
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 the IEEE software using `fscanf`.

```
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 IEEE workspace.

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

ASCII Write Properties

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

The EOI line can also be asserted when a terminator is written to the instrument. The terminator is defined by the `EOSCharCode` property. When `EOSMode` is configured to `write` or `read&write`, the EOI line is asserted when the `EOSCharCode` property value is written to the instrument.

All occurrences of `\n` in the command written to the instrument are replaced with the `EOSCharCode` property value if `EOSMode` is set to `write` or `read&write`.

Reading and Writing Binary Data

These functions are used when reading and writing binary data:

Function	Purpose
fread	Read binary data from an instrument.
fwrite	Write binary data to an instrument.

These properties are associated with reading and writing binary data:

Property	Purpose
ValuesReceived	Specifies the total number of values read from the instrument.
ValuesSent	Specifies the total number of values sent to the instrument.
InputBufferSize	Specifies the total number of bytes that can be queued in the input buffer at one time.
OutputBufferSize	Specifies the total number of bytes that can be queued in the output buffer at one time.
EOSMode	Configures the End-Of-String termination mode.
EOSCharCode	Specifies the End-Of-String terminator.

You use the fwrite function to write binary data to an instrument.

By default, the fwrite function operates in a synchronous mode. This means that fwrite blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the Timeout property

By default the fwrite function writes binary data using the uchar precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for fwrite.

You use the fread function to read binary data from the instrument.

The fread function blocks the MATLAB command line until one of the following occurs:

- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled
- The specified number of values is read
- The EOI line is asserted
- The terminator is received as specified by the `EOSCharCode` property (if defined)

By default the `fread` function reads binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fread`.

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

The following example illustrates how you can download the TDS 210 oscilloscope screen display to the IEEE software. The screen display data is transferred to the IEEE software 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')
```

4-23

```
fprintf(g,'HARDCOPY START')
```

Asynchronously transfer the data from the instrument to the input buffer.

```
readasync(g)
```

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

```
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 IEEE workspace.

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

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

Display the image.

```
imagesc(a)
```

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

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

The resulting bitmap image is shown below.
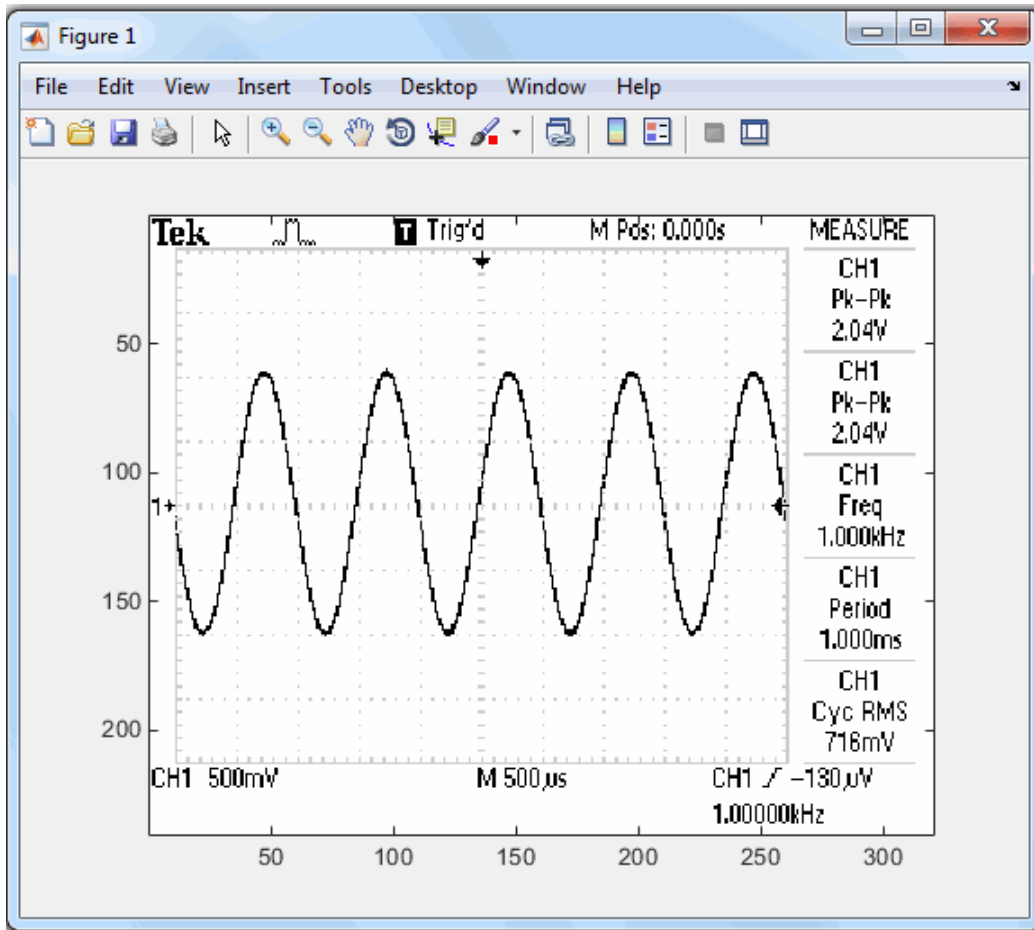
Parsing Input Data Using scanstr

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.

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

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

```
fopen(g)
```

3 **Write and read data** — 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'
```

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
```

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

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

```
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.EOIMode = 'off';
g.EOSMode = 'write';
fprintf(g,'*IDN?')
out = fscanf(g)
out =

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.EOSMode = 'none';
fprintf(g,'*IDN?')
out = fscanf(g)

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.

```
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.

```
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.

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

Events and Callbacks

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 MATLAB 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 callback function as the value for the associated callback property.

This example uses the 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',O,1);
g.BytesAvailableFcnMode = 'eosCharCode';
g.BytesAvailableFcn = @instrcallback;
fopen(g)
fprintf(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.

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

You can see the code for the built-in `instrcallback` function by using the `type` command.

Event Types and Callback Properties

The GPIB event types and associated callback properties are described below.

GPIB Event Types and Callback Properties

Event Type	Associated Property Name
Bytes-available	`BytesAvailableFcn`
	`BytesAvailableFcnCount`
	`BytesAvailableFcnMode`
Error	`ErrorFcn`
Output-empty	`OutputEmptyFcn`
Timer	`TimerFcn`
	`TimerPeriod`

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.

Responding To Event Information

You can respond to event information in a callback function or in a record file. Event information stored in a callback function uses two fields: `Type` and `Data`. The `Type` field contains the event type, while the `Data` field contains event-specific information. 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 16-6 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

Event Type	Field	Field Value
Bytes available	Type	BytesAvailable

Event Type	Field	Field Value
	`Data.AbsTime`	day-month-year hour:minute:second
Error	`Type`	`Error`
	`Data.AbsTime`	day-month-year hour:minute:second
	`Data.Message`	An error string
Output empty	`Type`	`OutputEmpty`
	`Data.AbsTime`	day-month-year hour:minute:second
Timer	`Type`	`Timer`
	`Data.AbsTime`	day-month-year hour:minute:second

The `Data` field values are described below.

AbsTime Field

`AbsTime` is defined for all events, and indicates the absolute time the event occurred. The absolute time is returned using the MATLAB `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 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. Note that if you are executing a local callback function from within a 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'};
```

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. 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,O);
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.

Using Events and Callbacks to Read Binary Data

This example extends "Reading and Writing Binary Data" on page 4-22 by using the 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.

```
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.

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

Initiate the asynchronous read operation, and begin generating events.

```
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 the MATLAB workspace as unsigned 8-bit integers.

```
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.

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

Triggers

In this section...
"Using the trigger Function" on page 4-36
"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.

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.

    ```
    fprintf(g,'BM:NCycles 25000')
    ```

```
fprintf(g,'Trigger:Source Bus')
fprintf(g,'BM:State On')
```

Trigger the instrument.

```
trigger(g)
```

Disable the burst mode.

```
fprintf(g,'BM:State Off')
```

While the function generator is triggered, the sine wave is saved to the Ref A memory location of the oscilloscope. The saved waveform is shown below.
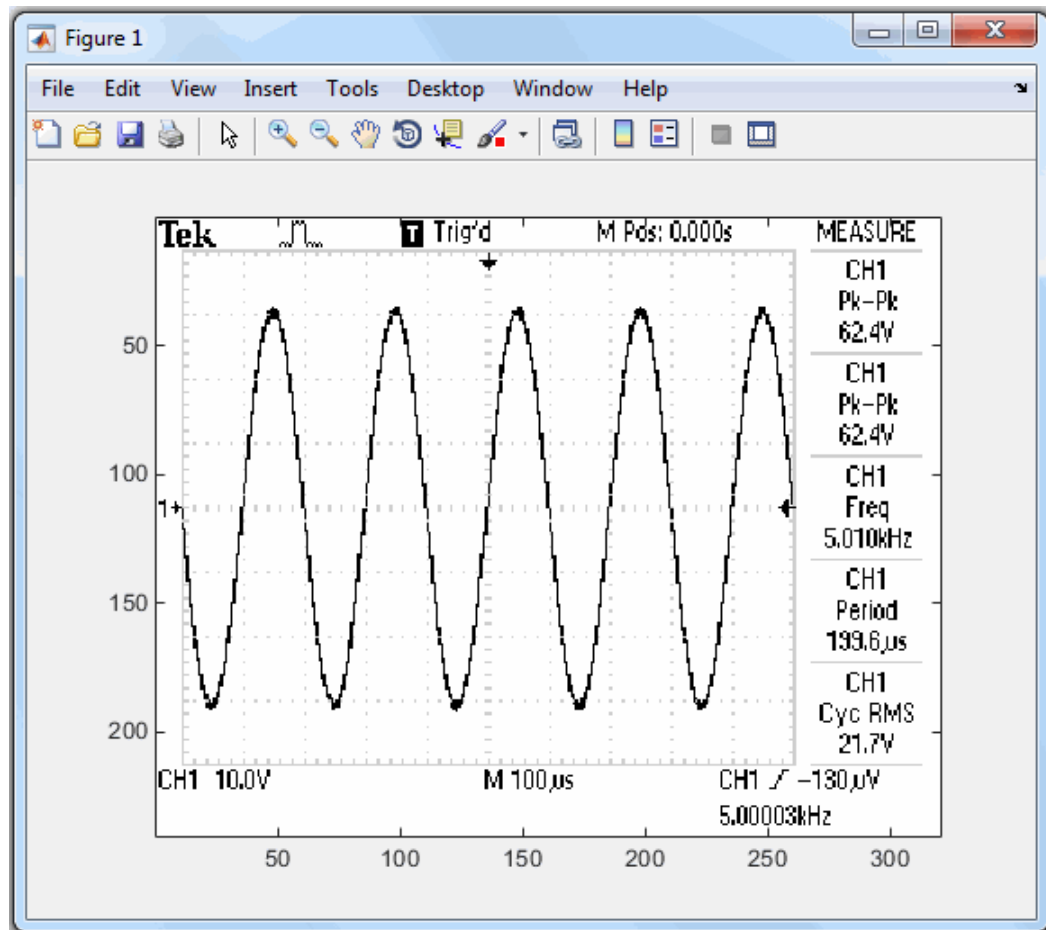
4. **Disconnect and clean up** — When you no longer need g, you should disconnect it from the instrument, and remove it from memory and from the MATLAB workspace.

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

Serial Polls

In this section...
"Using the spoll Function" on page 4-39
"Executing a Serial Poll" on page 4-39

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 4-8 for more information on the GPIB bus lines and the RQS bit.

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 4-11:

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

2 **Connect to the instrument** — Connect `g1` to the function generator and connect `g2` to the oscilloscope.

```
fopen([g1 g2])
```

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

```
g1.Timeout = 1;
g2.Timeout = 1;
```

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

```
fprintf(g1,'*CLS');
fprintf(g1,'*ESE 32');
fprintf(g1,'*SRE 32');
```

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

```
fprintf(g2,'*CLS')
fprintf(g2,'*PSC 0')
fprintf(g2,'*ESE 32')
fprintf(g2,'DESE 32')
fprintf(g2,'*SRE 32')
```

Determine if any instrument needs servicing.

```
spoll([g1 g2])
ans =
[]
```

Query the voltage value for each instrument.

```
fprintf(g1,'Volt?')
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]
ans =
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.

volt2 =
''
```

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
```

5

Controlling Instruments Using VISA

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

VISA Overview

In this section...
"What Is VISA?" on page 5-2
"Interfaces Used with VISA" on page 5-2
"Supported Vendor and Resource Names" 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.

The Instrument Control Toolbox software supports the GPIB, VXI, GPIB-VXI, TCP/IP using VXI-11, TCP/IP using HiSLIP, USB, RSIB, 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 VXI and PXI Interfaces" on page 5-9
- "Working with the GPIB-VXI Interface" on page 5-21
- "Working with the Serial Port Interface" on page 5-26

- "Working with the USB Interface" on page 5-30
- "Working with the TCP/IP Interface for VXI-11 and HiSLIP" on page 5-34
- "Working with the RSIB Interface" on page 5-38

If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to the Getting Started documentation, linked to at the top of the Instrument Control Toolbox Doc Center page.

Supported Vendor and Resource Names

When you use instrhwinfo to find commands to configure the interface objects, you must use valid vendor or resource names. The supported values for *vendor* are given below.

Vendor	Description
agilent	Agilent Technologies VISA
ni	National Instruments VISA
tek	Tektronix VISA (see note below for 64-bit support)

Note: For 64-bit Tektronix VISA support, it is important to note the following if you have a multi-vendor VISA installation (e.g., you have installed drivers from Tektronix and another vendor such as Agilent). If you are using 64-bit Tektronix VISA on a machine with VISA implementations from multiple vendors, it is required that Tektronix VISA be configured as the primary VISA for it to be usable with Instrument Control Toolbox. Most 64-bit VISA implementations include a utility that allows you to select the primary and preferred VISA implementations. Use the VISA utility to set Tektronix VISA to be the primary VISA implementation on your machine. This step can be accomplished at any time, regardless of the order of installation of the VISA drivers.

The format for rsrc name is given below for the supported VISA interfaces. The values indicated by brackets are optional. You can use the instrument's VISA Alias for rsrcname.

Interface	Resource Name
GPIB	GPIB[board]::primary_address[::secondary_address]::INSTR
GPIB-VXI	GPIB-VXI[chassis]::VXI_logical_address::INSTR

Interface	Resource Name
RSIB	RSIB::remote_host::INSTR (provided by NI VISA only)
Serial	ASRL[port_number]::INSTR
TCPIP (VXI-11)	TCPIP[board]::remote_host[::inst0]::INSTR
TCPIP (HiSLIP)	TCPIP[board]::remote_host[::hislip0]::INSTR
USB	USB[board]::manid::model_code::serial_No[::interface_No]::INSTR
VXI	VXI[chassis]::VXI_logical_address::INSTR

The `rsrcname` parameters are described below.

Parameter	Description
`board`	Board index (optional — defaults to 0)
`chassis`	VXI chassis index (optional — defaults to 0)
`interface_No`	USB interface
`lan_device_name`	Local Area Network (LAN) device name (optional — defaults to inst0)
`manid`	Manufacturer ID of the USB instrument
`model_code`	Model code for the USB instrument
`port_number`	Serial port number (optional — defaults to 1)
`primary_address`	Primary address of the GPIB instrument
`remote_host`	Host name or IP address of the instrument
`secondary_address`	Secondary address of the GPIB instrument (optional — defaults to 0)
`serial_No`	Index of the instrument on the USB hub
`VXI_logical_address`	Logical address of the VXI instrument

`obj = visa('`*vendor*`','rsrcname','`*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.

Working with the GPIB Interface

In this section...
"Understanding VISA-GPIB" on page 5-5
"Creating a VISA-GPIB Object" on page 5-5
"VISA-GPIB Address" on page 5-7

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 "GPIB Overview" on page 4-2 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 "Connecting to the Instrument" on page 2-4, you can also configure properties during object creation.

Before you create a VISA object, you must find the instrument in the appropriate vendor VISA explorer. When you find the instrument configured, note its VISA resource string and create the object using that information.

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 = visa('ni','GPIB0::1::0::INSTR');
```

The VISA-GPIB object `vg` now exists in the MATLAB workspace.

To open a connection to the instrument type:

```
fopen (vg);
```

You can then display the class of `vg` with the `whos` command.

```
whos vg
  Name       Size          Bytes  Class

   vg        1x1             884  visa object

Grand total is 16 elements using 884 bytes
```

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

VISA-GPIB Descriptive Properties

Property Name	Description
Name	Specify a descriptive name for the VISA-GPIB object.
RsrcName	Indicate the resource name for a VISA instrument.
Type	Indicate the object type.

You can display the values of these properties for `vg`.

```
vg.Name
ans =

'VISA-GPIB0-1'

vg.RsrcName
```

```
ans =

'GPIB0::1::0::INSTR'

vg.Type
ans =

'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
```

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

Property Name	Description
BoardIndex	Specify the index number of the GPIB board.
PrimaryAddress	Specify the primary address of the GPIB instrument.
SecondaryAddress	Specify the secondary address of the GPIB instrument.

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 .

```
vg.BoardIndex

ans =

     0

vg.PrimaryAddress

ans =

     1

vg.SecondaryAddress

ans =

     0
```

Working with VXI and PXI Interfaces

In this section...
"Understanding VISA-VXI " on page 5-9
"Understanding VISA-PXI" on page 5-10
"Creating a VISA-VXI Object" on page 5-10
"VISA-VXI Address" on page 5-12
"Register-Based Communication" on page 5-13

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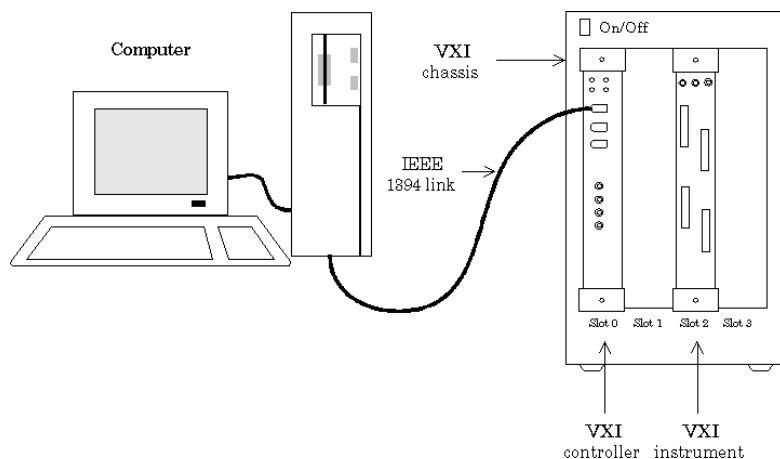
The VXI interface is supported through a VISA-VXI object. Many of the features associated with a VISA-VXI object are similar to the features associated with other instrument objects. Therefore, only functions and properties that are unique to VISA's VXI interface are discussed in this section.

Refer to "GPIB Overview" on page 4-2 to learn about general toolbox capabilities such as writing and reading text and binary data, using events and callbacks, and so on.

Understanding VISA-PXI

A PXI interface is supported through a VISA-PXI object. Features associated with a VISA-PXI object are identical to the features associated with a VISA-VXI object. Information provided for working with VISA-VXI in this section also works for VISA-PXI.

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

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.

Before you create a VISA object, you must find the instrument in the appropriate vendor VISA explorer. When you find the instrument configured, note the resource string and create the object using that information. 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 = visa('agilent','VXI0::32::INSTR');
```

The VISA-VXI object `vv` now exists in the MATLAB workspace.

To open a connection to the instrument, type:

```
fopen (vv);
```

You can then display the class of `vv` with the `whos` command.

```
whos vv
```

```
  Name       Size         Bytes  Class

  vv         1x1            882  visa object

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

Property Name	Description
Name	Specify a descriptive name for the VISA-VXI object.
RsrcName	Indicate the resource name for a VISA instrument.
Type	Indicate the object type.

You can display the values of these properties for vv.

```
vg.Name
ans =

'VISA-VXI0-32'

vg.RsrcName
ans =

'VXI0::32::INSTR'

vg.Type
ans =

'visa-vxi'
```

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.

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-VXI object `vv` is given below.

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

Communication Address
   ChassisIndex:      0
   LogicalAddress:    32

Communication State
   Status:            closed
   RecordStatus:      off

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

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

Property Name	Description
ChassisIndex	Indicate the index number of the VXI chassis.
LogicalAddress	Specify the logical address of the VXI instrument.
Slot	Indicate the slot location of the VXI instrument.

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.

```
fopen(vv)
vv.ChassisIndex
ans =


0

vv.LogicalAddress
ans =

32

vv.Slot
ans =

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 "Using High-Level Memory Functions" on page 5-16 for more information. The main advantage of

the low-level functions is speed. Refer to "Using Low-Level Memory Functions" on page 5-18 for more information.

The functions associated with register-based write and read operations are as follows.

VISA-VXI Register-Based Write and Read Functions

Function Name	Description
memmap	Map memory for low-level memory read and write operations.
mempeek	Low-level memory read from the VXI register.
mempoke	Low-level memory write to the VXI register.
memread	High-level memory read from the VXI register.
memunmap	Unmap memory for low-level memory read and write operations.
memwrite	High-level memory write to the VXI register.

The properties associated with register-based write and read operations are given below.

VISA-VXI Register-Based Write and Read Properties

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

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.

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

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

```
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.

```
vv.MemorySpace
ans =
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`.

```
vv.MemoryBase
ans =
    200000H
```

```
vv.MemorySize
ans =
    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.

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

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

Register	Offset	Description
ID	0	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.
Device Type	2	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.
Status	4	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.
Offset	6	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.

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.

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

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

```
fopen(vv)
```

3 **Write and read data** — The following command performs a high-level read of the ID Register, which has an offset of 0.

```
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.

```
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.

```
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.

```
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.

```
dec2bin(reg4,16)
ans =
1011000000000000
```

Restore the original register value, which is stored in the `reg24` variable.

```
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.

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

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 "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.

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

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

```
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 space.

```
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.

```
vv.MappedMemoryBase
```

```
ans =
    16737610H

vv.MappedMemorySize
ans =
    16
```

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

```
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.

```
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.

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

Restore the original register value.

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

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

```
memunmap(vv)
vv.MappedMemoryBase
ans =
    0H

vv.MappedMemorySize
ans =
    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 5-21
"Creating a VISA-GPIB-VXI Object" on page 5-22
"VISA-GPIB-VXI Address" on page 5-24

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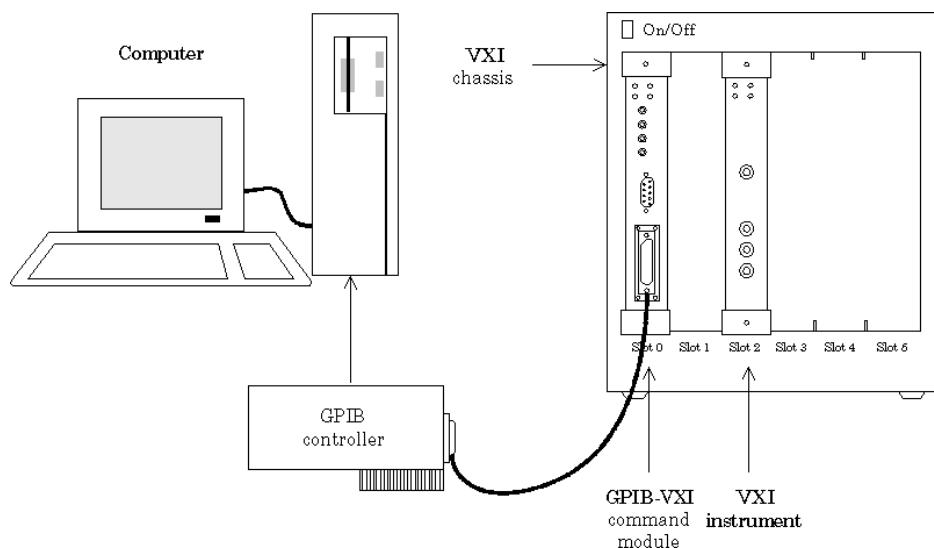
The GPIB-VXI interface is supported through a VISA-GPIB-VXI object. The features associated with a VISA-GPIB-VXI object are similar to the features associated with GPIB and VISA-VXI objects. Therefore, only functions and properties that are unique to VISA's GPIB-VXI interface are discussed in this section.

Refer to "GPIB Overview" on page 4-2 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 5-13 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.

Before you create a VISA object, you must find the instrument in the appropriate vendor VISA explorer. When you find the instrument configured, note the resource string and create the object using that information. 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,

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

The VISA-GPIB-VXI object `vgv` now exists in the MATLAB workspace.

To open a connection to the instrument type:

```
fopen (vgv);
```

You can then display the class of `vgv` with the `whos` command.

```
whos vgv
  Name      Size          Bytes  Class
```

```
vgv       1x1            892  visa object
```

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

Property Name	Description
Name	Specify a descriptive name for the VISA-GPIB-VXI object.
RsrcName	Indicate the resource name for a VISA instrument.
Type	Indicate the object type.

You can display the values of these properties for vgv.

```
vgv.Name
ans =
VISA-GPIB-VXI0-80

vgv.RsrcName
ans =
GPIB-VXI0::80::INSTR

vgv.Type
ans =
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.

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.

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-VXI object vgv is given below.

```
VISA-GPIB-VXI Object Using AGILENT Adaptor : VISA-GPIB-VXI0-80

Communication Address
   ChassisIndex:      0
   LogicalAddress:    80

Communication State
   Status:            closed
   RecordStatus:      off

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

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

Property Name	Description
BoardIndex	Indicate the index number of the GPIB board.
ChassisIndex	Specify the index number of the VXI chassis.
LogicalAddress	Specify the logical address of the VXI instrument.
PrimaryAddress	Indicate the primary address of the GPIB-VXI command module.
SecondaryAddress	Indicate the secondary address of the VXI instrument.
Slot	Indicate the slot location of the VXI instrument.

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 5-22.

```
fopen(vgv)
get(vgv,{'BoardIndex','ChassisIndex','LogicalAddress',...
'PrimaryAddress','SecondaryAddress','Slot'})
ans =
    [0]    [0]    [80]    [9]    [10]    [2]
```

Working with the Serial Port Interface

In this section...

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 "Serial Port Overview" on page 6-2 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.

Some vendors do not provide VISA serial support until you enable a port in their configuration tools. Before you create a VISA object, find the instrument in the appropriate vendor VISA explorer. When you find the instrument configured, note the resource string and create the object using that information. For example, to create

a VISA-serial object that is associated with the COM1 port, and that uses National Instruments VISA,

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

The VISA-serial object vs now exists in the MATLAB workspace.

To open a connection with the instrument, type:

```
fopen (vs);
```

You can then display the class of vs with the whos command.

```
whos vs
  Name       Size          Bytes  Class

  vs         1x1             888  visa object

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

Property Name	Description
Name	Specify a descriptive name for the VISA-serial object.
Port	Indicate the serial port name.
RsrcName	Indicate the resource name for a VISA instrument.
Type	Indicate the object type.

You can display the values of these properties for vs.

```
vs.Name
ans =
VISA-Serial-ASRL1

vs.Port
ans =
ASRL1

vs.RsrcName
```

```
ans =
ASRL1::INSTR

vs.Type
ans =
visa-serial
```

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.

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-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 6-8. These properties are given below.

VISA-Serial Communication Properties

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

Refer to your instrument documentation for an explanation of its supported communication settings. 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 5-26.

```
vs.BaudRate
ans =
    9600

vs.DataBits
ans =
    8

vs.Parity
ans =
    none

vs.StopBits
ans =
    1

vs.Terminator
ans =
   LF
```

Working with the USB Interface

In this section...
"Creating a VISA-USB Object" on page 5-30
"VISA-USB Address" on page 5-32

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.

Before you create a VISA object, you must find the instrument in the appropriate vendor VISA explorer. When you find the instrument configured, note the resource string and create the object using that information. For example, to create a VISA-USB object that uses National Instruments VISA,

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

The VISA-USB object `vu` now exists in the MATLAB workspace.

To open a connection to the instrument, type:

```
fopen (vu);
```

You can display the class of `vu` with the `whos` command.

```
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

Property Name	Description
Name	Specify a descriptive name for the VISA-USB object.
RsrcName	Indicate the resource name for a VISA instrument.
Type	Indicate the object type.

You can display the values of these properties for vs.

```
vu.Name
ans =
VISA-USB-0-0x1234-125-A22-5-0

vu.RsrcName
ans =
USB::0x1234::125::A22-5::INSTR

vu.Type
ans =
visa-usb
```

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.

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-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
```

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

Property Name	Description
BoardIndex	Specify the index number of the USB board in VISA configuration (optional — defaults to 0).
InterfaceIndex	Specify the USB interface number (optional).
ManufacturerID	Specify the manufacturer ID of the USB instrument.
ModelCode	Specify the model code of the USB instrument.
SerialNumber	Specify the index of the USB instrument on the USB hub.

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

You can display the address property values for the VISA-USB object vu, created in "Creating a VISA-USB Object" on page 5-30.

```
fopen(vu)
vu.ManufacturerID
ans =
    0x1234

vu.ModelCode
ans =
    125

vu.SerialNumber
ans =
    A22-5
```

Working with the TCP/IP Interface for VXI-11 and HiSLIP

In this section...
"Understanding VISA-TCP/IP" on page 5-34
"Creating a VISA-TCPIP Object" on page 5-34
"VISA-TCPIP Address" on page 5-36

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. Both VXI-11 and HiSLIP protocols are supported.

Refer to "TCP/IP and UDP Overview" on page 7-2 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 TCP/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.

Before you create a VISA object, you must find the instrument in the appropriate vendor VISA explorer. When you find the instrument configured, note the resource string and create the object using that information. For example, to create a VISA-TCPIP object that uses National Instruments VISA associated with an instrument at IP address 216.148.60.170 using the VXI-11 protocol,

```
vt = visa('ni','TCPIP::216.148.60.170::INSTR');
```

The VISA-TCPIP object `vt` now exists in the MATLAB workspace.

To open an connection to the instrument, type:

```
fopen (vt);
```

You can display the class of `vt` with the `whos` command.

```
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

Property Name	Description
Name	Specify a descriptive name for the VISA-TCPIP object.
RsrcName	Indicate the resource name for a VISA instrument.
Type	Indicate the object type.

You can display the values of these properties for `vt`.

```
vt.Name
ans =
VISA-TCPIP-216.148.60.170

vt.RsrcName
ans =
TCPIP::216.148.60.170::INSTR

vt.Type
ans =
visa-tcpip
```

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.

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-TCPIP object `vt` 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
```

VISA-TCPIP Address

The VISA-TCPIP address consists of

- Board index (optional, from the VISA configuration)
- Remote host of the instrument
- The protocol, either VXI-11 or HiSLIP
- 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

Property Name	Description
BoardIndex	Specify the index number of the TCP/IP board in VISA configuration (optional — defaults to 0).

Property Name	Description
RemoteHost	Specify the remote host name or IP address of the instrument.
LANName	Specify the LAN device name of the instrument.

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

You can display the address property values for the VISA-TCPIP object vt, created in "Creating a VISA-TCPIP Object" on page 5-34.

```
fopen(vt)
vt.RemoteHost
ans =
    216.148.60.170
```

Working with the RSIB Interface

In this section...
"Understanding VISA-RSIB" on page 5-38
"Creating a VISA-RSIB Object" on page 5-38
"VISA-RSIB Address" on page 5-40

Understanding VISA-RSIB

RSIB Passport for VISA allows you to control and exchange data remotely with Rohde & Schwarz spectrum and network analyzers over a local area network. The RSIB interface is supported by National Instruments VISA only. It also requires the Rohde & Schwarz VISA passport. You can use MATLAB and Rohde & Schwarz spectrum and network analyzers to perform complex data analysis on measured telecommunication signals and to verify simulated data against real measurement data.

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.

Before you create a VISA object, you must find the instrument in the appropriate vendor VISA explorer. When you find the instrument configured, note the resource string and create the object using that information. 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.

To open a connection to the instrument, type:

```
fopen (vr);
```

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

Property Name	Description
Name	Specify a descriptive name for the VISA-RSIB object.
RsrcName	Indicate the resource name for a VISA instrument.
Type	Indicate the object type.

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

```
vr.Name
ans =
    VISA-RSIB0-192.168.1.33

vr.RsrcName
ans =
    RSIB0::192.168.1.33::INSTR

vr.Type
ans =
    visa-RSIB
```

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:    O
   ValuesReceived:    O
   ValuesSent:        O
```

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

Property Name	Description
RemoteHost	Specify the remote host name or IP address of the instrument

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

You can display the address property value for the VISA-RSIB object vr, created in "Creating a VISA-RSIB Object" on page 5-38.

```
fopen(vr)
vr.RemoteHost
ans =
    192.168.1.33
```

Working with the Generic Interface

In this section...
"Generic VISA" on page 5-42
"VISA Node and Generic VISA Support in Test & Measurement Tool" on page 5-42
"Generic VISA Support in the Command-line Interface" on page 5-42

Generic VISA

In both the command-line toolbox and the Test & Measurement Tool, a generic VISA interface is now supported. In the Test & Measurement Tool, generic devices will appear in the **More** node under the **VISA** node. In the command-line toolbox, they are available as a type 'generic'.

For example, if you have a generic VISA device that is made by National Instruments, you could use the instrhwinfo function to see it, as follows.

```
instrhwinfo('visa','ni','generic')
```

This generic support can be used to communicate over open VISA sockets, USB Raw, etc.

VISA Node and Generic VISA Support in Test & Measurement Tool

In the Test & Measurement Tool, instruments that use the VISA interface show up under the **VISA** node in the instrument tree. For example, if you are using a TCP/IP instrument with the VISA interface, instead of a **TCP/IP - VISA** node in the tree, you will see a **VISA** node, with a **TCP/IP** node under it. It is easier to see what protocols can be used with the VISA interface with the **VISA** node.

Generic devices will appear in a **More** node under the **VISA** node in the instrument tree. If your instrument is recognizable as a type such as 'gpib' or 'tcpip', it will show up in that type-specific node. For example, a TCP/IP instrument would show up in the **TCPIP** node under the **VISA** node. But if it is a generic instrument, it will show up in the **More** node.

Generic VISA Support in the Command-line Interface

You can use the instrhwinfo function to see generic VISA devices.

```
instrhwinfo('INTERFACE', 'ADAPTOR', 'TYPE')
```

INTERFACE is `'visa'`. ADAPTOR can be `'agilent'`, `'ni'` or `'tek'`, depending on whether your instrument vendor is Agilent, National Instruments, or Tektronix. TYPE can be `'gpib'`, `'vxi'`, `'gpib-vxi'`, `'serial'`, `'tcpip'`, `'usb'`, `'rsib'`, `'pxi'`, or `'generic'`. Use `'generic'` when it is a generic device or form of communication.

For example:

```
>> instrhwinfo('visa','ni','generic')

ans =

              AdaptorDllName: [1x73 char]
           AdaptorDllVersion: 'Version 3.0.0'
                 AdaptorName: 'NI'
            AvailableChassis: []
        AvailableSerialPorts: []
            InstalledBoardIds: {4x1 cell}
         ObjectConstructorName: {4x1 cell}
                  SerialPorts: []
               VendorDllName: 'visa32.dll'
       VendorDriverDescription: 'National Instruments VISA Driver'
          VendorDriverVersion: 4.1000
```

This shows that there are four generic devices using the NI adaptor. If you look at the object constructor names, you can see the four devices.

```
>> ans.ObjectConstructorName

ans =

      'visa('ni', 'TCPIP0::systemlinux.dhcp::7::SOCKET');'
      'visa('ni', 'USB0::0x3923::0x7166::01574E49::RAW');'
      'visa('ni', 'TCPIP0::172.31.146.177::4000::SOCKET');'
      'visa('ni', 'TCPIP0::a-d60541-000006.dhcp::5025::SOCKET');'
```

In this example, there are three instruments capable of TCP/IP socket communication, and one of raw USB communication.

To communicate with a generic instrument using the `generic` interface, use the same functions, properties, and work flows described in the other interface sections of the VISA documentation.

Note: Some VISA drivers do not support EOI Mode. Therefore, if a device does not support EOI Mode, the VISA `generic` adaptor will default to `'off'` for the EOI Mode property, so that it does not cause a failure.

Reading and Writing ASCII Data Using VISA

In this section...

This example explores ASCII read and write operations with a VISA object. The instrument used was a Tektronix® TDS 210 oscilloscope.

The VISA object supports seven interfaces: serial, GPIB, VXI, GPIB-VXI, TCPIP, USB, and RSIB. This example explores ASCII read and write operations using a VISA-GPIB object. However, ASCII read and write operations for VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-TCPIP, VISA-SERIAL, and VISA-USB objects are identical to each other. Therefore, you can use the same commands. The only difference is the resource name specified in the VISA constructor.

ASCII read and write operations for the VISA-serial object are identical to ASCII read and write operations for the serial port object. Therefore, to learn how to perform ASCII read and write operations for the VISA-serial object, you should refer to the Serial Port ASCII Read/Write tutorial.

ASCII read and write operations for the VISA-RSIB object are identical to the ASCII read and write operations for the VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-TCPIP, and VISA-USB objects, except the VISA-RSIB object does not support the EOSCharCode and EOSMode properties.

These functions are used when reading and writing text:

Function	Purpose
fprintf	Write text to an instrument.
fscanf	Read data from an instrument and format as text.

These properties are associated with reading and writing text:

Property	Purpose
`ValuesReceived`	Specifies the total number of values read from the instrument.
`ValuesSent`	Specifies the total number of values sent to the instrument.
`InputBufferSize`	Specifies the total number of bytes that can be queued in the input buffer at one time.
`OutputBufferSize`	Specifies the total number of bytes that can be queued in the output buffer at one time.
`EOSMode`	Configures the End-Of-String termination mode.
`EOSCharCode`	Specifies the End-Of-String terminator.
`EOIMode`	Enables or disables the assertion of the EOI mode at the end of a write operation.

Configuring and Connecting to the Instrument

You need to create a VISA-GPIB object. In this example, an object is created using the `ni` driver and the VISA resource string shown below.

```
v = visa('ni', 'GPIB0::2::INSTR');
```

Before you can perform a read or write operation, you must connect the VISA-GPIB object to the instrument with the `fopen` function.

```
fopen(v);
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
v.Status
ans =
    open
```

Writing ASCII Data

You use the `fprintf` function to write ASCII data to the instrument. For example, the `'Display:Contrast'` command will change the display contrast of the oscilloscope.

```
fprintf(v, 'Display:Contrast 45');
```

By default, the `fprintf` function operates in a synchronous mode. This means that `fprintf` blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the `Timeout` property

By default the `fprintf` function writes ASCII data using the `%s\n` format. You can also specify the format of the command written by providing a third input argument to `fprintf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s. For example:

```
fprintf(v, '%s', 'Display:Contrast 45');
```

ASCII Write Properties

OutputBufferSize

The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the instrument at once. By default, `OutputBufferSize` is `512`.

```
v.OutputBufferSize
ans =
    512
```

If the command specified in `fprintf` contains more than 512 bytes, an error is returned and no data is written to the instrument.

EOIMode, EOSMode, and EOSCharCode

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

All occurrences of `\n` in the command written to the instrument are replaced with the `EOSCharCode` property value if `EOSMode` is set to `write` or `read&write`.

ValuesSent

The `ValuesSent` property is updated by the number of values written to the instrument. Note that by default `EOSMode` is set to none. Therefore, `EOSCharCode` is not sent as the last byte of the write.

```
fprintf(v, 'Display:Contrast 45');
v.ValuesSent
ans =
    57
```

Clear any data in the input buffer before moving to the next step.

```
flushinput(v);
```

Reading ASCII Data

You use the `fscanf` function to read ASCII data from the instrument. For example, the oscilloscope command `'Display:Contrast?'` returns the oscilloscope's display contrast:

```
fprintf(v, 'Display:Contrast?');
data = fscanf(v)

data =

    45
```

`fscanf` blocks until one of the following occurs:

- The EOI line is asserted
- The terminator is received as specified by the `EOSCharCode` property
- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled
- The specified number of values is received

By default, the `fscanf` function reads data using the `'%c'` format. You can also specify the format of the data read by providing a second input argument to `fscanf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s. For example, the following command will return the voltage as a decimal:

```
fprintf(v, 'Display:Contrast?');
data = fscanf(v, '%d')

data =

    45
```

```
isnumeric(data)

ans =

    1
```

ASCII Read Properties

InputBufferSize

The `InputBufferSize` property specifies the maximum number of bytes you can read from the instrument. By default, `InputBufferSize` is `512`.

```
v.InputBufferSize
ans =
    512
```

ValuesReceived

The `ValuesReceived` property indicates the total number of values read from the instrument. Note the last value received is a linefeed.

```
fprintf(v, 'Display:Contrast?');
data = fscanf(v)

data =

    45

v.ValuesReceived

ans =

     9
```

EOSMode and EOSCharCode

To terminate the data transfer based on receiving `EOSCharCode`, you should set the `EOSMode` property to `read` or `read&write` and the `EOSCharCode` property to the ASCII code for which the read operation should terminate. For example, if you set `EOSMode` to `read` and `EOSCharCode` to `10`, then one of the ways that the read terminates is when the linefeed character is received.

The standard response to the vertical gain query is in scientific notation.

```
fprintf(v, 'CH1:Scale?')
data = fscanf(v)

data =

    1.0E0
```

Now configure the VISA-GPIB object to terminate the read operation when the 'E' character is received. The first read terminates when the 'E' character is received.

```
set(v, 'EOSMode', 'read')
set(v, 'EOSCharCode', double('E'))
fprintf(v, 'CH1:Scale?')
data = fscanf(v)

data =

1.0E
```

If you perform a second read operation, it terminates when the EOI line is asserted.

```
data = fscanf(v)

data =

    0
```

Cleanup

If you are finished with the VISA-GPIB object, disconnect it from the instrument, remove it from memory, and remove it from the workspace.

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

Reading and Writing Binary Data Using VISA

In this section...
"Configuring and Connecting to the Instrument" on page 5-52
"Writing Binary Data" on page 5-52
"Binary Write Properties" on page 5-53
"Reading Binary Data" on page 5-54
"Binary Read Properties" on page 5-54
"Cleanup" on page 5-56

This example explores binary read and write operations with a VISA object. The instrument used was a Tektronix® TDS 210 oscilloscope.

This tutorial explores binary read and write operations using a VISA-GPIB object. However, binary read and write operations for VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-TCPIP, and VISA-USB objects are identical to each other. Therefore, you can use the same commands. The only difference is the resource name specified in the VISA constructor.

Binary read and write operations for the VISA-serial object are identical to binary read and write operations for the serial port object. Therefore, to learn how to perform binary read and write operations for the VISA-serial object, you should refer to the Serial Port Binary Read/Write tutorial.

Binary read and write operations for the VISA-RSIB object are identical to the binary read and write operations for the VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-TCPIP, and VISA-USB objects, except the VISA-RSIB object does not support the `EOSCharCode` and `EOSMode` properties.

These functions are used when reading and writing binary data:

Function	Purpose
`fread`	Read binary data from the instrument.
`fwrite`	Write binary data to the instrument.

These properties are associated with reading and writing binary data:

Property	Purpose
ValuesReceived	Specifies the total number of values read from the instrument.
ValuesSent	Specifies the total number of values sent to the instrument.
InputBufferSize	Specifies the total number of bytes that can be queued in the input buffer at one time.
OutputBufferSize	Specifies the total number of bytes that can be queued in the output buffer at one time.
EOSMode	Configures the End-Of-String termination mode.
EOSCharCode	Specifies the End-Of-String terminator.

Configuring and Connecting to the Instrument

You need to create a VISA-GPIB object. In this example, an object is created using the `ni` driver and the VISA resource string shown below.

```
v = visa('ni', 'GPIB0::2::INSTR');
```

Before you can perform a read or write operation, you must connect the VISA-GPIB object to the instrument with the `fopen` function.

```
fopen(v);
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
v.Status
ans =
    open
```

Writing Binary Data

You use the `fwrite` function to write binary data to the instrument. For example, the following command will send a sine wave to the instrument. By default, the `fwrite` function operates in a synchronous mode. This means that `fwrite` blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the `Timeout` property

By default the `fwrite` function writes binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fwrite`.

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

Binary Write Properties

OutputBufferSize

The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the instrument at once. By default, `OutputBufferSize` is 512.

```
v.OutputBufferSize
ans =
    512
```

Configure the object's output buffer size to 3000. Note the `OutputBufferSize` can be configured only when the object is not connected to the instrument.

```
fclose(v);
v.OutputBufferSize = 3000;
fopen(v);
```

Writing Int16 Binary Data

Now write a waveform as an int16 array.

```
fprintf(v, 'Data:Destination RefB');
fprintf(v, 'Data:Encdg SRPbinary');
fprintf(v, 'Data:Width 2');
fprintf(v, 'Data:Start 1');

t = (0:499) .* 8 * pi / 500;
data = round(sin(t) * 90 + 127);
fprintf(v, 'CURVE #3500');
```

Note that one int16 value consists of two bytes. Therefore, the following command will write 1000 bytes.

```
fwrite(v, data, 'int16')
```

ValuesSent

The `ValuesSent` property indicates the total number of values written to the instrument since the object was connected to the instrument.

```
v.ValuesSent
ans =
    576
```

Reading Binary Data

You use the `fread` function to read binary data from the instrument.

By default, the `fread` function reads data using the uchar precision and blocks the MATLAB command line until one of the following occurs:

- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled
- The specified number of values is read
- The EOI line is asserted
- The terminator is received as specified by the `EOSCharCode` property (if defined)

By default the `fread` function reads binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fread`.

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

Binary Read Properties

InputBufferSize

The `InputBufferSize` property specifies the maximum number of bytes you can read from the instrument. By default, `InputBufferSize` is 512.

```
v.InputBufferSize
ans =
    512
```

Configure the object's input buffer size to 5100. Note the `InputBufferSize` can be configured only when the object is not connected to the instrument.

```
fclose(v);
v.InputBufferSize = 5100;
fopen(v);
```

Reading Int16 Binary Data

Now read the same waveform on channel 1 as an int16 array.

```
fprintf(v, 'Data:Source CH1');
fprintf(v, 'Data:Encdg SRIbinary');
fprintf(v, 'Data:Width 2');
fprintf(v, 'Data:Start 1');
fprintf(v, 'Curve?')
```

Note that one int16 value consists of two bytes. Therefore, the following command will read 2400 bytes.

```
data = fread(v, 1200, 'int16');
```

ValuesReceived

The `ValuesReceived` property indicates the total number of values read from the instrument.

```
v.ValuesReceived
```

```
ans =

    1200
```

Since we may not have read all of the values, clear the input buffer.

```
flushinput(v);
```

EOSMode and EOSCharCode

For VISA-GPIB, objects, the terminator is defined by setting the object's `EOSMode` property to read and setting the object's `EOSCharCode` property to the ASCII code for

the character received. For example, if the **EOSMode** property is set to read and the **EOSCharCode** property is set to 10, then one of the ways that the read terminates is when the linefeed character is received.

Configure the GPIB object's terminator to the letter E.

```
set(v, 'EOSMode', 'read');
set(v, 'EOSCharCode', double('E'));
```

Now, read the channel 1's signal frequency.

```
fprintf(v, 'Measurement:Meas1:Source CH1')
fprintf(v, 'Measurement:Meas1:Type Freq')
fprintf(v, 'Measurement:Meas1:Value?')
```

Note: that the first read terminates due to the **EOSCharCode** being detected, while the second read terminates due to the EOI line being asserted.

```
data = fread(v, 30);
char(data)'

Warning: The EOI line was asserted or the EOSCharCode was detected
% before SIZE values were available.

ans =

    9.9E

data = fread(v, 30);
char(data)'

Warning: The EOI line was asserted or the EOSCharCode was detected
% before SIZE values were available.

ans =

    37
```

Cleanup

If you are finished with the VISA-GPIB object, disconnect it from the instrument, remove it from memory, and remove it from the workspace.

```
fclose(v);
```

```
delete(v);
clear v
```

Asynchronous Read and Write Operations Using VISA

In this section...
"Functions and Properties" on page 5-58
"Synchronous Versus Asynchronous Operations" on page 5-59
"Configuring and Connecting to the Instrument" on page 5-59
"Reading Data Asynchronously" on page 5-60
"Asynchronous Read Properties" on page 5-60
"Using Callbacks During an Asynchronous Read" on page 5-61
"Writing Data Asynchronously" on page 5-62
"Cleanup" on page 5-62

This example explores asynchronous read and write operations using a VISA-GPIB object. The instrument used was a Tektronix® TDS 2024 oscilloscope.

This tutorial explores asynchronous read and write operations for a VISA-GPIB-VXI object. However, asynchronous read and write operations for VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-TCPIP, and VISA-USB objects are identical to each other. Therefore, you can use the same commands. The only difference is the resource name specified in the VISA constructor.

Asynchronous read and write operations for the VISA-serial object are identical to asynchronous read and write operations for the serial port object. Therefore, to learn how to perform asynchronous read and write operations for the VISA-serial object, you should refer to the Serial Port Asynchronous Read/Write tutorial.

Asynchronous read and write operations are not supported for the VISA-RSIB object.

Functions and Properties

These functions are associated with reading and writing text asynchronously:

Function	Purpose
fprintf	Write text to a instrument.
readasync	Asynchronously read bytes from an instrument.
stopasync	Stop an asynchronous read or write operation.

These properties are associated with reading and writing text asynchronously:

Property	Purpose
BytesAvailable	Indicates the number of bytes available in the input buffer.
TransferStatus	Indicates what type of asynchronous operation is in progress.

Additionally, you can use all the callback properties during asynchronous read and write operations.

Synchronous Versus Asynchronous Operations

The VISA object can operate in either synchronous or asynchronous mode. In synchronous mode, the MATLAB command line is blocked until

- The read or write operation completes
- A timeout occurs as specified by the Timeout property

In asynchronous mode, control is immediately returned to the MATLAB command line. Additionally, you can use callback properties and callback functions to perform tasks as data is being written or read. For example, you can create a callback function that notifies you when the read or write operation has finished.

Configuring and Connecting to the Instrument

You need to create a VISA-GPIB object. In this example, an object is created using the ni driver and the VISA resource string shown below.

```
v = visa('ni', 'GPIB0::2::INSTR')
```

Before you can perform a read or write operation, you must connect the VISA-GPIB object to the instrument with the fopen function.

```
fopen(v)
```

If the object was successfully connected, its Status property is automatically configured to open.

```
v.Status
ans =
    open
```

Reading Data Asynchronously

The VISA-GPIB object's asynchronous read functionality is controlled with the `readasync` function. Query the instrument for the channel 1 vertical scale:

```
fprintf(v, 'CH1:Scale?');
```

The `readasync` function can asynchronously read the data from the instrument. The `readasync` function returns control to the MATLAB command prompt immediately.

```
readasync(v, 20);
```

The `readasyn` function without a size specified will assume size is given by the difference between the `InputBufferSize` property value and the `BytesAvailable` property value. In the above example, size is 20. The asynchronous read terminates when one of the following occurs:

- The terminator is read as specified by the `EOSCharCode` property
- The specified number of bytes are stored in the input buffer
- A timeout occurs as specified by the `Timeout` property
- The EOI line has been asserted

An error event will be generated if `readasync` terminates due to a timeout.

Asynchronous Read Properties

Transfer Status

The `TransferStatus` property indicates what type of asynchronous operation is in progress. For VISA-GPIB objects, `TransferStatus` can be configured as `read`, `write`, or `idle`.

```
v.TransferStatus
ans =
    idle
```

While an asynchronous read is in progress, an error occurs if you execute another write or asynchronous read operation. You can stop the asynchronous read operation with the `stopasync` function. The data in the input buffer will remain after `stopasync` is called. This allows you to bring the data that was read into the MATLAB workspace with one of the synchronous read routines (`fscanf`, `fgetl`, `fgets`, or `fread`).

BytesAvailable

If we now look at the `BytesAvailable` property, you see that 6 bytes were read.

```
v.BytesAvailable
ans =
     6
```

You can bring the data into the MATLAB workspace with the `fscanf` function.

```
data = fscanf(v, '%g')
data =
      1
```

Using Callbacks During an Asynchronous Read

Now, configure the VISA-GPIB object to notify you when a line feed has been read. The `BytesAvailableFcnMode` property controls when the `BytesAvailable` event is created. By default, the `BytesAvailable` event is created when the `EOSCharCode` character is received. The `BytesAvailable` event can also be created after a certain number of bytes have been read. Note that the `BytesAvailableFcnMode` property cannot be configured while the object is connected to the instrument.

```
set(v, 'BytesAvailableFcn', {'dispcallback'});
set(v, 'EOSCharCode', 10);
```

The callback function `dispcallback` displays a message containing the type of the event, the name of the object that caused the event to occur, and the time the event occurred.

Now, query the instrument for the frequency of the signal. Once the linefeed has been read from the instrument and placed in the input buffer, `dispcallback` will be executed and a message will be displayed to the MATLAB command window indicating that a `BytesAvailable` event occurred.

```
fprintf(v, 'CH2:Scale?');
readasync(v);
```

Allow time for a response. In a typical application this is where you could do other tasks.

```
pause(0.5);
```

```
A BytesAvailable event occurred for VISA-GPIB0-2 at 01-Jun-2005 15:08:34.
```

```
v.BytesAvailable

ans =

     6

data = fscanf(v, '%c', 6)

data =

    à (0
```

Note that the last value read is the line feed (10):

```
real(data)

ans =

   224    32    40    16    48    10
```

Writing Data Asynchronously

You can perform an asynchronous write with the `fprintf` or `fwrite` functions by passing an `'async'` flag as the last input argument.

While an asynchronous write is in progress, an error occurs if you execute a read or write operation. You can stop an asynchronous write operation with the `stopasync` function. The data remaining in the output buffer will be flushed.

Also configure the object to notify you when the write operation has completed by defining an asynchronous write callback.

```
set(v, 'OutputEmptyFcn', {'dispcallback'});
fprintf(v, 'CH1:Scale?', 'async');
```

Cleanup

If you are finished with the VISA-GPIB object, disconnect it from the instrument, remove it from memory, and remove it from the workspace.

```
fclose(v);
delete(v);
```

```
clear v
```

Controlling Instruments Using the Serial Port

This chapter describes specific issues related to controlling instruments that use the serial port.

Serial Port Overview

In this section...
"What Is Serial Communication?" on page 6-2
"Serial Port Interface Standard" on page 6-2
"Supported Platforms" on page 6-3
"Connecting Two Devices with a Serial Cable" on page 6-3
"Serial Port Signals and Pin Assignments" on page 6-4
"Serial Data Format" on page 6-8
"Finding Serial Port Information for Your Platform" on page 6-11

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 "Serial Port Object" on page 6-14. If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to the Getting Started documentation that is linked to at the top of the main Instrument Control Toolbox Doc Center page.

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-232" 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

- Linux® 32-bit
- Linux 64-bit
- Mac OS X
- Mac OS X 64-bit
- Microsoft® Windows 32-bit
- Microsoft Windows 64-bit

Connecting Two Devices with a Serial Cable

The RS-232 and RS-485 standard defines the two devices connected with a serial cable as the Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE). This terminology reflects the RS-232 origin as a standard for communication between a computer terminal and a modem.

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 6-4 for more information about serial port pins.
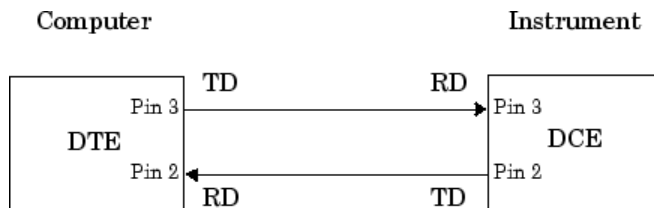
If you connect two DTEs or two DCEs using a straight serial cable, then the TD pin on each device is connected to the other, and the RD pin on each device is connected to the other. Therefore, to connect two like devices, you must use a *null modem* cable. As shown below, null modem cables cross the transmit and receive lines in the cable.
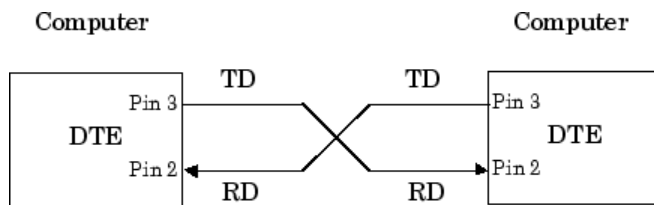
Note You can connect multiple RS-422 or RS-485 devices to a serial port. If you have an RS-232/RS-485 adaptor, then you can use the serial port object with these devices.

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.
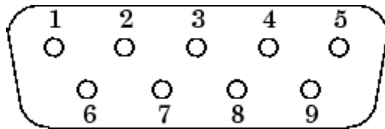
The pins and signals associated with the 9-pin connector are described below. Refer to the RS-232 or the RS-485 standard for a description of the signals and pin assignments used for a 25-pin connector.

Serial Port Pin and Signal Assignments

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

The term "data set" is synonymous with "modem" or "device," while the term "data terminal" is synonymous with "computer."

Note The serial port pin and signal assignments are with respect to the DTE. For example, data is transmitted from the TD pin of the DTE to the RD pin of the DCE.

Signal States

Signals can be in either an *active state* or an *inactive state*. An active state corresponds to the binary value 1, while an inactive state corresponds to the binary value 0. An active

signal state is often described as *logic 1*, *on*, *true*, or a *mark*. An inactive signal state is often described as *logic 0*, *off*, *false*, or a *space*.

For data signals, the "on" state occurs when the received signal voltage is more negative than -3 volts, while the "off" state occurs for voltages more positive than 3 volts. For control signals, the "on" state occurs when the received signal voltage is more positive than 3 volts, while the "off" state occurs for voltages more negative than -3 volts. The voltage between -3 volts and +3 volts is considered a transition region, and the signal state is undefined.

To bring the signal to the "on" state, the controlling device *unasserts* (or *lowers*) the value for data pins and *asserts* (or *raises*) the value for control pins. Conversely, to bring the signal to the "off" state, the controlling device asserts the value for data pins and unasserts the value for control pins.

The "on" and "off" states for a data signal and for a control signal are shown below.
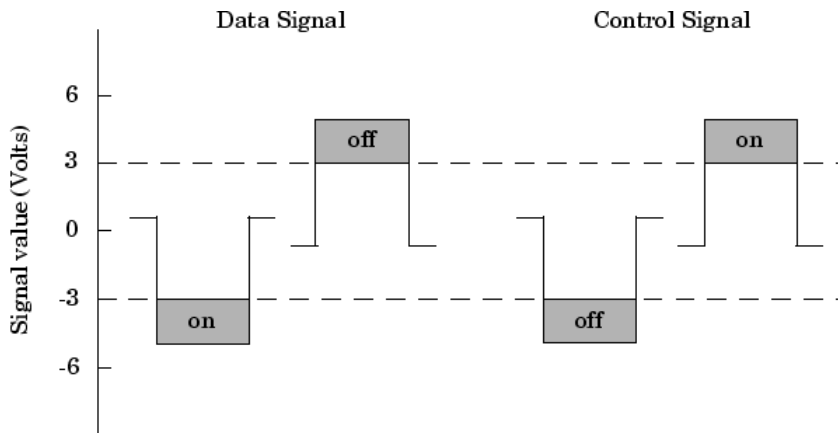
The Data Pins

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 6-7
- "The DTR and DSR Pins" on page 6-7
- "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 6-42.

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.

Start bit   Data bits   Parity bit   Stop bits
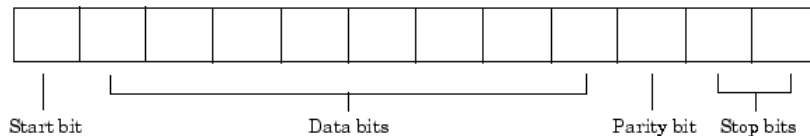
The format for serial port data is often expressed using the following notation: number of data bits - parity type - number of stop bits

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 6-18.

Synchronous and Asynchronous Communication

The RS-232 and the RS-485 standard support two types of communication protocols: synchronous and asynchronous.

Using the synchronous protocol, all transmitted bits are synchronized to a common clock signal. The two devices initially synchronize themselves to each other, and then continually send characters to stay synchronized. Even when actual data is not really being sent, a constant flow of bits allows each device to know where the other is at any given time. That is, each bit that is sent is either actual data or an idle character. Synchronous communications allows faster data transfer rates than asynchronous methods, because additional bits to mark the beginning and end of each data byte are not required.

Using the asynchronous protocol, each device uses its own internal clock resulting in bytes that are transferred at arbitrary times. So, instead of using time as a way to synchronize the bits, the data format is used.

In particular, the data transmission is synchronized using the start bit of the word, while one or more stop bits indicate the end of the word. The requirement to send these additional bits causes asynchronous communications to be slightly slower than synchronous. However, it has the advantage that the processor does not have to deal with the additional idle characters. Most serial ports operate asynchronously.

Note When used in this guide, the terms "synchronous" and "asynchronous" refer to whether read or write operations block access to the MATLAB Command Window.

How Are the Bits Transmitted?

By definition, serial data is transmitted one bit at a time. The order in which the bits are transmitted follows these steps:

1 The start bit is transmitted with a value of 0.
2 The data bits are transmitted. The first data bit corresponds to the least significant bit (LSB), while the last data bit corresponds to the most significant bit (MSB).
3 The parity bit (if defined) is transmitted.
4 One or two stop bits are transmitted, each with a value of 1.

The number of bits transferred per second is given by the *baud rate*. The transferred bits include the start bit, the data bits, the parity bit (if defined), and the stop bits.

Start and Stop Bits

As described in "Synchronous and Asynchronous Communication" on page 6-9, 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

Parity Type	Description
Even	The data bits plus the parity bit produce an even number of 1s.
Mark	The parity bit is always 1.
Odd	The data bits plus the parity bit produce an odd number of 1s.
Space	The parity bit is always 0.

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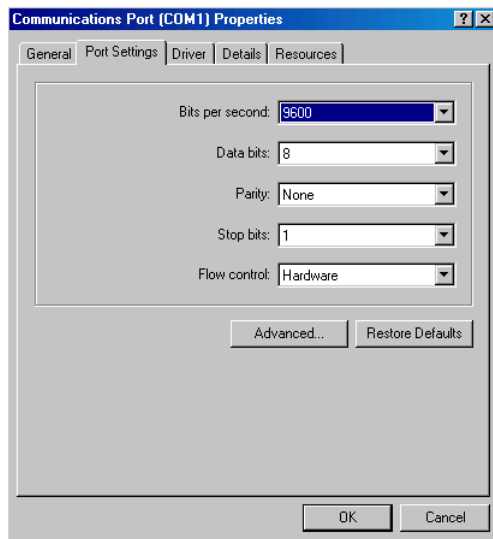
UNIX Platform

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,

```
setserial -bg /dev/ttyS*

/dev/ttyS0 at 0x03f8 (irq = 4) is a 16550A
/dev/ttyS1 at 0x02f8 (irq = 3) is a 16550A
```

To display detailed information about `ttyS0`,

```
setserial -ag /dev/ttyS0

/dev/ttyS0, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4
        Baud_base: 115200, close_delay: 50, divisor: 0
        closing_wait: 3000, closing_wait2: infinte
        Flags: spd_normal skip_test session_lockout
```

Note If the `setserial -ag` command does not work, make sure that you have read and write permission for the port.

For all supported UNIX platforms, including Mac OS X, you can use the `stty` command to display or configure serial port information. For example, to display serial port properties for `ttyS0`, type:

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

To configure the baud rate to 4800 bits per second, type:

```
stty speed 4800 < /dev/ttyS0 > /dev/ttyS0
```

Note: This is an example of setting `tty` parameters, not the baud rate. To set baud rate using MATLAB serial interface refer to "Configuring Communication Settings" on page 6-17.

Serial Port Object

In this section...
"Creating a Serial Port Object" on page 6-14
"Serial Port Object Display" on page 6-16

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 a serial port enter

```
s = serial('port');
```

This 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 device. '*port*' object name will depend upon the platform that the serial port is on.

```
 instrhwinfo('serial')
```

provides a list of available serial ports. This list is an example of serial constructors on different platforms:

Platform	Serial Constructor
Linux 32 and 64-bit	serial('/dev/ttyS0');
Mac OS X and Mac OS X 64-bit	serial('/dev/tty.KeySerial1');
Microsoft Windows 32 and 64-bit	serial('com1');

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

```
whos s
```

```
  Name      Size           Bytes  Class

  s         1x1              512  serial object

Grand total is 11 elements using 512 bytes
```

> **Note:** The first time you try to access a serial port in MATLAB using the `s = serial('com1')` call, make sure that the port is free and is not already open in any other application. If the port is open in another application, MATLAB cannot access it. Once you have accessed in MATLAB, you can open the same port in other applications and MATLAB will continue to use it along with any other application that has it open as well.

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

Property Name	Description
Name	Specify a descriptive name for the serial port object.
Port	Indicate the platform-specific serial port name.
Type	Indicate the object type.

You can display the values of these properties for `s`

`s.Name`

```
 ans =
    'Serial-COM1'
```

`s.Port`

```
 ans =
    'COM1'
```

`s.Type`

```
 ans =
    'serial'
```

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

Serial Port Object Display

The serial port object provides a convenient display that summarizes important configuration and state information. You can invoke the display summary 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** on a Windows machine is given here.

```
s = serial('COM1')

Serial Port Object : Serial-COM1

Communication Settings
   Port:              COM1
   BaudRate:          9600
   Terminator:        'LF'

Communication State
   Status:            closed
   RecordStatus:      off

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

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 6-8. These properties are as follows.

Serial Port Communication Properties

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

Caution 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 "Serial Port Object" on page 6-14 .

```
get(s,{'BaudRate','DataBits','Parity','StopBits','Terminator'})

ans =
[9600]    [8]    'none'    [1]    'LF'
```

Writing and Reading Data

In this section...
"Asynchronous Write and Read Operations" on page 6-18
"Rules for Completing Write and Read Operations" on page 6-24
"Writing and Reading Text Data" on page 6-25
"Writing and Reading Binary Data" on page 6-29

Asynchronous Write and Read Operations

These functions are associated with reading and writing text asynchronously:

Function	Purpose
fprintf	Write text to an instrument.
readasync	Asynchronously read bytes from an instrument.
stopasync	Stop an asynchronous read or write operation.

These properties are associated with reading and writing text asynchronously:

Property	Purpose
BytesAvailable	Indicates the number of bytes available in the input buffer.
TransferStatus	Indicates what type of asynchronous operation is in progress.
ReadAsyncMode	Indicates whether data is read continuously in the background or whether you must call the readasync function to read data asynchronously.

Additionally, you can use all the callback properties during asynchronous read and write operations.

Asynchronous write and read operations do not block access to the MATLAB Command Window. 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 6-34 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 3-16.

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 the MATLAB workspace, you use one of the synchronous (blocking) read functions such as `fgetl`, `fgets`, `fscanf`, or `fread`. If data is available in the input buffer, these functions will return quickly.

Note: This example is Windows specific.

```
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 the MATLAB workspace.

```
s.ReadAsyncMode = 'manual';
fprintf(s,'*IDN?')
```

```
s.BytesAvailable
ans =
     0
readasync(s)
s.BytesAvailable
ans =
     56
out = fscanf(s);
```

Writing and Reading Asynchronous Data

This example explores asynchronous read and write operations using a serial port object. The instrument used was a Tektronix(R) TDS 210 oscilloscope.

To begin, create a serial port object associated with the COM1 port. The oscilloscope is configured to a baud rate of 9600, 1 stop bit, a line feed terminator, no parity, and no flow control.

```
s = serial('COM1');
s.Baudrate=9600;
s.StopBits=1;
s.Terminator='LF';
s.Parity='none';
s.FlowControl='none';
```

Before you can perform a read or write operation, you must connect the serial port object to the instrument with the fopen function.

```
fopen(s);
```

If the object was successfully connected, its Status property is automatically configured to open.

```
s.Status

ans =
    open
```

To begin, read data continuously.

```
s.ReadAsyncMode='continuous';
```

Now, query the instrument for the peak-to-peak value of the signal on channel 1.

```
fprintf(s, 'Measurement:Meas1:Source CH1');
```

```
fprintf(s, 'Measurement:Meas1:Type Pk2Pk');
fprintf(s, 'Measurement:Meas1:Value?');
```

Allow time for a response. In a typical application this is where you could do other tasks.

```
pause(0.5);
```

Since the `ReadAsyncMode` property is set to `continuous`, the object is continuously asking the instrument if any data is available. Once the last `fprintf` function completes, the instrument begins sending data; the data is read from the instrument and is stored in the input buffer.

```
s.BytesAvailable
```

```
ans =
    14
```

You can bring the data from the object's input buffer into the MATLAB workspace with `fscanf`.

```
data = fscanf(s)
```

```
data =
    5.99999987E-2
```

Next, read the data manually.

```
s.ReadAsyncMode='manual';
```

Now, query the instrument for the frequency of the signal on channel 1.

```
fprintf(s, 'Measurement:Meas2:Source CH1');
fprintf(s, 'Measurement:Meas2:Type Freq');
fprintf(s, 'Measurement:Meas2:Value?');
```

Allow time for a response. In a typical application this is where you could do other tasks.

```
pause(0.5);
```

Once the last `fprintf` function completes, the instrument begins sending data. However, since `ReadAsyncMode` is set to `manual`, the object is not reading the data being sent from the instrument. Therefore, no data is being read and placed in the input buffer.

```
s.BytesAvailable
```

```
ans =
```

```
     0
```

Read the data.

```
readasync(s);
```

Allow time for a response.

```
pause(0.5);
```

It is important to remember that when the serial port object is in manual mode (the `ReadAsyncMode` property is configured to `manual`), data that is sent from the instrument to the computer is not automatically stored in the input buffer of the connected serial port object. Data is not stored until `readasync` or one of the blocking read functions is called.

Manual mode should be used when a stream of data is being sent from your instrument and you only want to capture portions of the data.

Defining an Asynchronous Read Callback

Continuing the example from the previous section, configure the serial object to notify you when a terminator has been read.

```
s.ReadAsyncMode='continuous';
s.BytesAvailableFcn={'dispcallback'};
```

Note, the default value for the `BytesAvailableFcnMode` property indicates that the callback function defined by the `BytesAvailableFcn` property will be executed when the terminator has been read.

```
s.BytesAvailableFcnMode
```

```
ans =
    terminator
```

The `dispcallback` function displays a message containing the type of the event, the name of the object that caused the event to occur, and the time the event occurred.

```
callbackTime = datestr(datenum(event.Data.AbsTime));
fprintf(['A ' event.Type ' event occurred for ' obj.Name ' at '
    callbackTime '.\n']);
```

Query the instrument for the period of the signal on channel 1. Once the terminator is read from the instrument and placed in the input buffer, `dispcallback` is

executed and a message is posted to the MATLAB command window indicating that a `BytesAvailable` event occurred.

```
fprintf(s, 'Measurement:Meas3:Source CH1')
fprintf(s, 'Measurement:Meas3:Type Period')
fprintf(s, 'Measurement:Meas3:Value?')
```

Allow time for a response.

```
pause(0.5);

A BytesAvailable event occurred for Serial-COM1 at <date and time>.

s.BytesAvailable

ans =
    7

data = fscanf(s, '%c', 10)

data =

    2.0E-6
```

Note that the last value read is the line feed (10).

Now suppose that halfway through the asynchronous read operation, you realize that the signal displayed on the oscilloscope was incorrect. Rather than waiting for the asynchronous operation to complete, you can use the `stopasync` function to stop the asynchronous read. Note that if an asynchronous write was in progress, the asynchronous write operation would also be stopped.

```
s.BytesAvailableFcn='';
fprintf(s, 'Curve?');
pause(0.25);
S.BytesAvailable


ans =
    126


stopasync(s);
s.BytesAvailable
```

```
ans =
    262
```

The data that has been read from the instrument remains in the input buffer. You can use one of the synchronous read functions to bring this data into the MATLAB workspace. However, since this data represents the wrong signal, the `flushinput` function is called to remove all data from the input buffer.

```
flushinput(s);
s.BytesAvailable
```

```
ans =
    0
```

You can perform an asynchronous write with the `fprintf` or `fwrite` functions by passing `'async'` as the last input argument.

```
fprintf(s, 'Measurement:Meas3:Value?', 'async')
```

If you are finished with the serial port object, disconnect it from the instrument, remove it from memory, and remove it from the workspace.

```
fclose(s);
delete(s);
clear 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.

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.

Note: Set the terminator property to ' ' (null), if appropriate, to ensure efficient throughput of binary data.

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

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.

These functions are used when reading and writing text:

Function	Purpose
fprintf	Write text to an instrument.
fscanf	Read data from an instrument and format as text.

These properties are associated with reading and writing text:

Property	Purpose
ValuesReceived	Specifies the total number of values read from the instrument.
ValuesSent	Specifies the total number of values sent to the instrument.
InputBufferSize	Specifies the total number of bytes that can be queued in the input buffer at one time.
OutputBufferSize	Specifies the total number of bytes that can be queued in the output buffer at one time.
Terminator	Character used to terminate commands sent to the instrument.

Note: This example is Windows specific.

---

1 **Create a serial port object** — Create the serial port object s associated with serial port COM1.

```
s = 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.

```
fprintf(s,'*IDN?')
s.BytesAvailable
ans =
    56
idn = fscanf(s)
idn =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

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

```
fprintf(s,'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(s)
source =
CH1
```

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

```
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.

```
fprintf(s,'MEASUREMENT:MEAS1:TYPE PK2PK')
fprintf(s,'MEASUREMENT:MEAS1:VALUE?')
```

Transfer data from the input buffer to the MATLAB workspace using `fscanf`.

```
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.

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

Usage Notes for Writing ASCII Data

By default, the `fprintf` function operates in a synchronous mode. This means that `fprintf` blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the `Timeout` property

By default the `fprintf` function writes ASCII data using the `%s\n` format. All occurrences of `\n` in the command being written to the instrument are replaced with the `Terminator` property value. When using the default format, `%s\n`, all commands written to the instrument will end with the `Terminator` character.

For the previous command, the linefeed (LF) is sent after `'Hello World 123'` is written to the instrument, thereby indicating the end of the command.

You can also specify the format of the command written by providing a third input argument to `fprintf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s.

ASCII Write Properties

The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the instrument at once. By default, `OutputBufferSize` is `512`.

```
s.OutputBufferSize
```

```
ans =
    512
```

The `ValuesSent` property indicates the total number of values written to the instrument since the object was connected to the instrument.

```
s.ValuesSent
```

```
ans =
    40
```

Usage Notes for Reading ASCII Data

By default, the `fscanf` function reads data using the `'%c'` format and blocks the MATLAB command line until one of the following occurs:

- The terminator is received as specified by the `Terminator` property
- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled
- The specified number of values is read

You can also specify the format of the data read by providing a second input argument to `fscanf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s.

ASCII Read Properties

The `InputBufferSize` property specifies the maximum number of bytes you can read from the instrument. By default, `InputBufferSize` is `512`.

```
s.InputBufferSize
```

```
ans =
    512
```

The `ValuesReceived` property indicates the total number of values read from the instrument, including the terminator.

```
s.ValuesReceived
```

```
ans =
    6
```

Writing and Reading Binary Data

This example explores binary read and write operations with a serial port object. The instrument used was a Tektronix® TDS 210 oscilloscope.

Functions and Properties

These functions are used when reading and writing binary data:

Function	Purpose
`fread`	Read binary data from the instrument.
`fwrite`	Write binary data to the instrument.

These properties are associated with reading and writing binary data:

Property	Purpose
`ValuesReceived`	Specifies the total number of values read from the instrument.
`ValuesSent`	Specifies the total number of values sent to the instrument.
`InputBufferSize`	Specifies the total number of bytes that can be queued in the input buffer at one time.
`OutputBufferSize`	Specifies the total number of bytes that can be queued in the output buffer at one time.

Configuring and Connecting to the Serial Object

You need to create a serial object. In this example, create a serial port object associated with the COM1 port.

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

Before you can perform a read or write operation, you must connect the serial port object to the instrument with the `fopen` function.

```
fopen(s)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
s.Status
```

```
ans =
    open
```

Writing Binary Data

You use the `fwrite` function to write binary data to the instrument. By default, the `fwrite` function operates in a synchronous mode. This means that `fwrite` blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the `Timeout` property

By default the `fwrite` function writes binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fwrite`.

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

Binary Write Properties

The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the instrument at once. By default, `OutputBufferSize` is `512`.

```
s.OutputBufferSize
```

```
ans =
    512
```

If the command specified in `fwrite` contains more than 512 bytes, an error is returned and no data is written to the instrument.

Configure the object's output buffer size to 3000. Note, the `OutputBufferSize` can be configured only when the object is not connected to the instrument.

```
fclose(s);
s.OutputBufferSize = 3000;
fopen(s);
```

The `ValuesSent` property indicates the total number of values written to the instrument since the object was connected to the instrument.

```
s.ValuesSent
```

```
ans =
    581
```

Writing Int16 Binary Data

Write a waveform as an `int16` array.

```
fwrite(s, 'Data:Destination RefB');
fwrite(s, 'Data:Encdg SRPbinary');
fwrite(s, 'Data:Width 2');
fwrite(s, 'Data:Start 1');

t = (0:499) .* 8 * pi / 500;
data = round(sin(t) * 90 + 127);
fwrite(s, 'CURVE #3500');
```

Note that one `int16` value consists of two bytes. Therefore, the following command will write 1000 bytes.

```
fwrite(s, data, 'int16')
```

Reading Binary Data

You use the `fread` function to read binary data from the instrument.

The `fread` function blocks the MATLAB command line until one of the following occurs:

- A timeout occurs as specified by the `Timeout` property
- The specified number of values is read
- The input buffer is filled

By default the `fread` function reads binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fread`.

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

Binary Read Properties

The `InputBufferSize` property specifies the maximum number of bytes that can be read from the instrument at once. By default, `InputBufferSize` is `512`.

```
s.InputBufferSize
```

```
ans =
    512
```

The `ValuesReceived` property indicates the total number of values read from the instrument.

```
s.ValuesReceived
```

```
ans =
    256
```

Reading int16 Binary Data

Read the same waveform on channel 1 as an `int16` array.

```
fread(s, 'Data:Source CH1');
fread(s, 'Data:Encdg SRPbinary');
fread(s, 'Data:Width 2');
fread(s, 'Data:Start 1');
fread(s, 'Data:Stop 2500');
```

```
fread(s, 'Curve?')
```

Note that one `int16` value consists of two bytes. Therefore, the following command will read 512 bytes.

```
data = fread(s, 256, 'int16'):
```

Cleanup

If you are finished with the serial port object, disconnect it from the instrument, remove it from memory, and remove it from the workspace.

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

Events and Callbacks

In this section...
"Event Types and Callback Properties" on page 6-34
"Responding To Event Information" on page 6-35
"Using Events and Callbacks" on page 6-37

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

Event Type	Associated Properties
Break interrupt	`BreakInterruptFcn`
Bytes available	`BytesAvailableFcn`
	`BytesAvailableFcnCount`
	`BytesAvailableFcnMode`
Error	`ErrorFcn`
Output empty	`OutputEmptyFcn`
Pin status	`PinStatusFcn`
Timer	`TimerFcn`
	`TimerPeriod`

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 4-30.

Note: You cannot use ASCII values larger than 127 characters with `fgetl`, `fgets`, or `BytesAvailableFnc`. The functions are limited to 127 binary characters.

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 6-4 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.

Responding To Event Information

You can respond to event information in a callback function or in a record file. Event information stored in a callback function uses two fields: `Type` and `Data`. The `Type` field contains the event type, while the `Data` field contains event-specific information. 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 16-6 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

Event Type	Field	Field Value
Break interrupt	`Type`	`BreakInterrupt`
	`Data.AbsTime`	day-month-year hour:minute:second
Bytes available	`Type`	`BytesAvailable`
	`Data.AbsTime`	day-month-year hour:minute:second

Event Type	Field	Field Value
Error	`Type`	`Error`
	`Data.AbsTime`	day-month-year hour:minute:second
	`Data.Message`	An error string
Output empty	`Type`	`OutputEmpty`
	`Data.AbsTime`	day-month-year hour:minute:second
Pin status	`Type`	`PinStatus`
	`Data.AbsTime`	day-month-year hour:minute:second
	`Data.Pin`	`CarrierDetect`, `ClearToSend`, `DataSetReady`, or `RingIndicator`
	`Data.PinValue`	`on` or `off`
Timer	`Type`	`Timer`
	`Data.AbsTime`	day-month-year hour:minute:second

The `Data` field values are as follows.

Field Name	Value
`AbsTime`	`AbsTime` is defined for all events, and indicates the absolute time the event occurred. The absolute time is returned using the MATLAB Command Window`clock` format.
`Pin`	`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 6-4 for a description of these pins.
`PinValue`	`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`.
`Message`	`Message` is used by the error event to store the descriptive message that is generated when an error occurs.

Using Events and Callbacks

This example uses the callback function `instrcallback` to display event-related information to the command line when a bytes-available event or an output-empty event occurs:

Note: This example is Windows specific.

---

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

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

2 **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;
```

3 **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)
```

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.

```
fprintf(s,'RS232?','async')
```

`instrcallback` is called after the `RS232?` command is sent, and when the terminator is read. The resulting displays are shown below.

```
OutputEmpty event occurred at 17:37:21 for the object:
Serial-COM1.

BytesAvailable event occurred at 17:37:21 for the object:
Serial-COM1.
```

5 Read the data from the input buffer.

```
out = fscanf(s)

out =
9600;0;0;NONE;LF
```
6 **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.

```
fclose(s)
delete(s)
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 4-29.

Note: You cannot use ASCII values larger than 127 characters with `fgetl`, `fgets`, or
`BytesAvailableFnc`. The functions are limited to 127 binary characters.

Using Control Pins

In this section...
"Control Pins" on page 6-39
"Signaling the Presence of Connected Devices" on page 6-39
"Controlling the Flow of Data: Handshaking" on page 6-42

Control Pins

As described in "Serial Port Signals and Pin Assignments" on page 6-4, 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

Property Name	Description
DataTerminalReady	Specify the state of the DTR pin.
FlowControl	Specify the data flow control method to use.
PinStatus	Indicate the state of the CD, CTS, DSR, and RI pins.
RequestToSend	Specify the state of the RTS pin.

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.

Connecting Two Modems

This example (shown on a Windows machine) 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.

```
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.

```
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 6-42
- "Software Handshaking" on page 6-43

Note Although you may be able to configure your device for both hardware handshaking
and software handshaking at the same time, the Instrument Control Toolbox software
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 6-7.

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 the Instrument Control Toolbox software, 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

Character	Integer Value	Description
Xon	17	Resume data transmission.
Xoff	19	Pause data transmission.

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.

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.

```
fprintf(s,'RS232:SOFTF ON')
s.FlowControl = 'software';
```

To pause data transfer, you write the numerical value 19 (Xoff) to the instrument.

```
fwrite(s,19)
```

To resume data transfer, you write the numerical value 17 (Xon) to the instrument.

```
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" on page 7-2
- "Creating a TCP/IP Object" on page 7-4
- "Creating a UDP Object" on page 7-9
- "Writing and Reading Data" on page 7-13
- "Events and Callbacks" on page 7-48
- "Using TCP/IP Server Sockets" on page 7-52

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

In this section...

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,

```
t = tcpip('sonytekawg.yourdomain.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 `whos` command.

```
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

Property Name	Description
`Name`	Specify a descriptive name for the TCP/IP object.
`RemoteHost`	Specify the remote host.
`RemotePort`	Specify the remote host port for the connection.
`Type`	Indicate the object type.

You can display the values of these properties for `t`.

```
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.yourdomain.com

Communication Settings
   RemotePort:      4000
   RemoteHost:      sonytekawg.yourdomain.com
   Terminator:      'LF'

Communication State
   Status:          closed
   RecordStatus:    off

Read/Write State
```

```
TransferStatus:    idle
BytesAvailable:    0
ValuesReceived:    0
ValuesSent:        0
```

Communicating with a Remote Host

In this example, you read a page from the RFC Editor Web site using a TCP/IP object:

1 Create and configure an instrument object — First you create a TCP/IP object in the MATLAB workspace. Port 80 is the standard port for Web servers.

```
t = tcpip('www.rfc-editor.org', 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.

```
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 a text file from the RFC Editor Web site using `'GET `*`(path/filename)`*`'`.

```
fprintf(t, 'GET /rfc/rfc793.txt');
```

The server receives the command and sends back the Web page. You can see if any data was sent back by looking at the `BytesAvailable` property of the object.

```
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.

```
while (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.

```
fclose(t);
delete(t);
clear t
```

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.yourdomain.com` and its port is 4000. The AWG's host IP address is 192.168.1.10 and is user configurable in the instrument. The associated host name is given by your network administrator. The port number is fixed and is found in the instrument's documentation.

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.

```
t = tcpip('sonytekawg.yourdomain.com', 4000);
```
2   Connect to the instrument — Connect to the remote instrument.

```
fopen(t)
```
3   Write and read data — Write a command to the instrument and read back the result.

```
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.

```
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.

```
t.Status
ans =
open
```

If you attempt to write to the instrument again, an error message is returned and the connection is automatically closed.

```
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.

```
fclose(t)
delete(t)
clear t
```

Creating a UDP Object

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.

Note: Although UDP is a stateless connection, opening a UDP object with an invalid host name will generate an error.

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 upd function. These general purpose properties provide information about the UDP object.

UDP Descriptive Properties

Property Name	Description
Name	Specify a descriptive name for the UDP object.
RemoteHost	Specify the remote host.
RemotePort	Specify the remote host port for the connection.
Type	Indicate the object type.
LocalPort	Specify the local host port, if you are expecting to receive data from the instrument.

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

```
u.Name
ans =
    UDP-127.0.0.1

u.RemoteHost
ans =
    127.0.0.1

u.RemotePort
ans =
    4012

u.Type
ans =
    udp

u.LocalPort'
ans =
    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
   Terminator:      'LF'

Communication State
   Status:          closed
   RecordStatus:    off

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

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:

   ```
   u1 = udp('doetom.dhpc', 'RemotePort', 8866, 'LocalPort', 8844)
   ```

   On host doetom.dhpc, create u2:

```
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`:

```
fopen(u1)
```

On host `doetom.dhpc`, open `u2`:

```
fopen(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`:

```
fprintf(u1, 'Ready for data transfer.')
```

On host `doetom.dhpc`, read data coming in from the remote host via `u2`:

```
fscanf(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.

```
fclose(u1)
delete(u1)
clear u1
```

When you no longer need `u2`, perform a similar cleanup on the host `doetom.dhpc`.

```
fclose(u2)
delete(u2)
clear u2
```

Writing and Reading Data

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 (`fscanf` and `readasync` only). For UDP objects, `DatagramTerminateMode` must be `off`.
- A datagram is received (for UDP objects, only when `DatagramTerminateMode` is `on`).

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

- The time specified by the `Timeout` property passes.
- The input buffer is filled.
- The specified number of values is read. For UDP objects, `DatagramTerminateMode` must be `off`.
- A datagram is received (for UDP objects, only when `DatagramTerminateMode` is `on`).

Note: Set the terminator property to `''` (null), if appropriate, to ensure efficient throughput of binary data.

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

Reading and Writing ASCII Data over TCP/IP

This section explores ASCII read and write operations with a TCP/IP object.

Note: Most bench-top instruments (oscilloscopes, function generators, etc.) that provide network connectivity do not use raw TCP socket communication for instrument command

and control. Instead, it is supported through the VISA standard. For more information on using VISA to communicate with your instrument, see "VISA Overview" on page 5-2.

Functions and Properties

These functions are used when reading and writing text:

Function	Purpose
fprintf	Write text to the server.
fscanf	Read data from the server and format as text.

These properties are associated with reading and writing text:

Property	Purpose
ValuesReceived	Specifies the total number of values read from the server.
ValuesSent	Specifies the total number of values sent to the server.
InputBufferSize	Specifies the total number of bytes that can be queued in the input buffer at one time.
OutputBufferSize	Specifies the total number of bytes that can be queued in the output buffer at one time.
Terminator	Character used to terminate commands sent to the server.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the TCP/IP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echotcpip('on', 4000)
```

You need to create a TCP/IP object. In this example, create a TCP/IP object associated with the host 127.0.0.1 (your local machine), port 4000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
t = tcpip('127.0.0.1', 4000);
```

Before you can perform a read or write operation, you must connect the TCP/IP object to the server with the `fopen` function.

```
fopen(t)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
t.Status
ans =
    open
```

Writing ASCII Data

You use the `fprintf` function to write ASCII data to the server.

```
fprintf(t, 'Hello World 123');
```

By default, the `fprintf` function operates in a synchronous mode. This means that `fprintf` blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the `Timeout` property

By default the `fprintf` function writes ASCII data using the `%s\n` format. All occurrences of `\n` in the command being written to the server are replaced with the `Terminator` property value. When using the default format, `%s\n`, all commands written to the server will end with the `Terminator` character.

For the previous command, the linefeed (LF) is sent after `'Hello World 123'` is written to the server, thereby indicating the end of the command.

You can also specify the format of the command written by providing a third input argument to `fprintf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s.

For example, the data command previously shown can be written to the server using three calls to `fprintf`.

```
fprintf(t, '%s', 'Hello');
fprintf(t, '%s', ' World');
fprintf(t, '%s\n', ' 123');
```

The `Terminator` character indicates the end of the command and is sent after the last call to `fprintf`.

ASCII Write Properties

The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the server at once. By default, `OutputBufferSize` is 512.

```
t.OutputBufferSize
ans =
    512
```

The `ValuesSent` property indicates the total number of values written to the server since the object was connected to the server.

```
t.ValuesSent
ans =
    32
```

Reading ASCII Data

You use the `fscanf` function to read ASCII data from the server. For example, to read back the data returned from the echo server for our first `fprintf` command:

```
data = fscanf(t)
data =
    Hello World 123
```

By default, the `fscanf` function reads data using the `'%c'` format and blocks the MATLAB command line until one of the following occurs:

- The terminator is received as specified by the `Terminator` property
- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled
- The specified number of values is read

You can also specify the format of the data read by providing a second input argument to `fscanf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s.

The following commands return a numeric value as a double.

Clear anything still in the input buffer from the previous commands.

```
flushinput(t);
```

Send the data to the server.

```
fprintf(t, '0.8000');
```

Read the response.

```
data = fscanf(t, '%f')
data =
    0.8000

isnumeric(data)
ans =
    1
```

ASCII Read Properties

The `InputBufferSize` property specifies the maximum number of bytes you can read from the server. By default, `InputBufferSize` is `512`.

```
t.InputBufferSize
ans =
    512
```

The `ValuesReceived` property indicates the total number of values read from the server, including the terminator.

```
t.ValuesReceived
ans =
    32
```

Cleanup

If you are finished with the TCP/IP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(t);
delete(t);
clear t

echotcpip('off');
```

Reading and Writing Binary Data over TCP/IP

This section explores binary read and write operations with a TCP/IP object.

> **Note:** Most bench-top instruments (oscilloscopes, function generators, etc.) that provide network connectivity do not use raw TCP socket communication for instrument command and control. Instead, it is supported through the VISA standard. For more information on using VISA to communicate with your instrument, see "VISA Overview" on page 5-2.

Functions and Properties

These functions are used when reading and writing binary data:

Function	Purpose
fread	Read binary data from the server.
fwrite	Write binary data to the server.

These properties are associated with reading and writing binary data:

Property	Purpose
ValuesReceived	Specifies the total number of values read from the server.
ValuesSent	Specifies the total number of values sent to the server.
InputBufferSize	Specifies the total number of bytes that can be queued in the input buffer at one time.
OutputBufferSize	Specifies the total number of bytes that can be queued in the output buffer at one time.
ByteOrder	Specifies the byte order of the server.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the TCP/IP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echotcpip('on', 4000)
```

You need to create a TCP/IP object. In this example, create a TCP/IP object associated with the host 127.0.0.1 (your local machine), port 4000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
t = tcpip('127.0.0.1', 4000);
```

You may need to configure the `OutputBufferSize` of the TCP/IP object. The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the server at once. By default, `OutputBufferSize` is 512.

```
t.OutputBufferSize
ans =
    512
```

If the command specified in `fwrite` contains more than 512 bytes, an error is returned and no data is written to the server. In this example 4000 bytes will be written to the server. Therefore, the `OutputBufferSize` is increased to 4000.

```
t.OutputBufferSize = 4000;
t.OutputBufferSize
ans =
    4000
```

You may need to configure the `ByteOrder` of the TCP/IP object. The `ByteOrder` property specifies the byte order of the server. By default `ByteOrder` is `bigEndian`.

```
t.ByteOrder
ans =
    bigEndian
```

If the server's byte order is little-endian, the `ByteOrder` property of the object can be configured to `littleEndian`:

```
t.ByteOrder = littleEndian
t.ByteOrder
ans =
    littleEndian
```

Before you can perform a read or write operation, you must connect the TCP/IP object to the server with the `fopen` function.

```
fopen(t)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
t.Status
ans =
    open
```

Writing Binary Data

You use the `fwrite` function to write binary data to the server. For example, the following command will send a sine wave to the server.

Construct the sine wave to be written to the server.

```
x = (0:999) .* 8 * pi / 1000;
data = sin(x);
```

Write the sine wave to the server.

```
fwrite(t, data, 'float32');
```

By default, the `fwrite` function operates in a synchronous mode. This means that `fwrite` blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the `Timeout` property

By default the `fwrite` function writes binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fwrite`.

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

Binary Write Properties

The `ValuesSent` property indicates the total number of values written to the server since the object was connected to the server.

```
t.ValuesSent
ans =
    1000
```

Configuring InputBufferSize

The `InputBufferSize` property specifies the maximum number of bytes that you can read from the server. By default, `InputBufferSize` is 512.

```
t.InputBufferSize
ans =
    512
```

Next, the waveform stored in the function generator's memory will be read. The waveform contains 4000 bytes. Configure the InputBufferSize to hold 4000 bytes. Note, the InputBufferSize can be configured only when the object is not connected to the server.

```
fclose(t);
t.InputBufferSize = 4000;
t.InputBufferSize
ans =
    4000
```

Now that the property is configured correctly, you can reopen the connection to the server:

```
fopen(t);
```

Reading Binary Data

You use the fread function to read binary data from the server.

The fread function blocks the MATLAB command line until one of the following occurs:

- A timeout occurs as specified by the Timeout property
- The specified number of values is read
- The InputBufferSize number of values is read

By default the fread function reads binary data using the uchar precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for fread.

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

For reading float32 binary data, send the waveform again. Closing the object clears any available data from earlier writes.

```
fwrite(t, data, 'float32');
```

Now read the same waveform as a `float32` array.

```
data = fread(t, 1000, 'float32');
```

The `ValuesReceived` property indicates the total number of values read from the server.

```
t.ValuesReceived
ans =
    1000
```

Cleanup

If you are finished with the TCP/IP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(t);
delete(t);
clear t

echotcpip('off');
```

Asynchronous Read and Write Operations over TCP/IP

This section explores Asynchronous read and write operations with a TCP/IP object.

Note: Most bench-top instruments (oscilloscopes, function generators, etc.) that provide network connectivity do not use raw TCP socket communication for instrument command and control. Instead, it is supported through the VISA standard. For more information on using VISA to communicate with your instrument, see "VISA Overview" on page 5-2.

Functions and Properties

These functions are associated with reading and writing text asynchronously:

Function	Purpose
fprintf	Write text to a server.
readasync	Asynchronously read bytes from a server.

Function	Purpose
stopasync	Stop an asynchronous read or write operation.

These properties are associated with reading and writing text asynchronously:

Property	Purpose
BytesAvailable	Indicates the number of bytes available in the input buffer.
TransferStatus	Indicates what type of asynchronous operation is in progress.
ReadAsyncMode	Indicates whether data is read continuously in the background or whether you must call the readasync function to read data asynchronously.

Additionally, you can use all the callback properties during asynchronous read and write operations.

Synchronous Versus Asynchronous Operations

The object can operate in synchronous mode or in asynchronous mode. When the object is operating synchronously, the read and write routines block the MATLAB command line until the operation has completed or a timeout occurs. When the object is operating asynchronously, the read and write routines return control immediately to the MATLAB command line.

Additionally, you can use callback properties and callback functions to perform tasks as data is being written or read. For example, you can create a callback function that notifies you when the read or write operation has finished.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the TCP/IP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echotcpip('on', 4000)
```

You need to create a TCP/IP object. In this example, create a TCP/IP object associated with the host 127.0.0.1 (your local machine), port 4000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
t = tcpip('127.0.0.1', 4000);
```

Before you can perform a read or write operation, you must connect the TCP/IP object to the server with the `fopen` function.

```
fopen(t)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
t.Status
ans =
    open
```

Reading Data Asynchronously

You can read data asynchronously with the TCP/IP object in one of these two ways:

- Continuously, by setting `ReadAsyncMode` to `continuous`. In this mode, data is automatically stored in the input buffer as it becomes available from the server.
- Manually, by setting `ReadAsyncMode` to `manual`. In this mode, you must call the `readasync` function to store data in the input buffer.

The `fscanf`, `fread`, `fgetl` and `fgets` functions are used to bring the data from the input buffer into MATLAB. These functions operate synchronously.

Reading Data Asynchronously – Continuous ReadAsyncMode

To begin, read data continuously.

```
t.ReadAsyncMode = continuous;
```

Now, send data to the server that will be returned for reading.

```
fprintf(t, 'Hello World 123');
```

Because the `ReadAsyncMode` property is set to `continuous`, the object is continuously checking whether any data is available. Once the last `fprintf` function completes, the server begins sending data, the data is read from the server and is stored in the input buffer.

```
t.BytesAvailable
ans =
    16
```

You can bring the data from the object's input buffer into the MATLAB workspace with `fscanf`.

```
fscanf(t)
ans =
    Hello World 123
```

Reading Data Asynchronously – Manual ReadAsyncMode

Next, read data manually.

```
t.ReadAsyncMode = manual;
```

Now, send data to the server that will be returned for reading.

```
fprintf(t, 'Hello World 456');
```

Once the last `fprintf` function completes, the server begins sending data. However, because `ReadAsyncMode` is set to `manual`, the object is not reading the data being sent from the server. Therefore no data is being read and placed in the input buffer.

```
t.BytesAvailable
ans =
    0
```

The `readasync` function can asynchronously read the data from the server. The `readasync` function returns control to the MATLAB command line immediately.

The `readasync` function takes two input arguments. The first argument is the server object and the second argument is the `size`, the amount of data to be read from the server.

The `readasync` function without a `size` specified assumes `size` is given by the difference between the `InputBufferSize` property value and the `BytesAvailable` property value. The asynchronous read terminates when:

- The terminator is read as specified by the `Terminator` property
- The specified number of bytes have been read
- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled

An error event will be generated if `readasync` terminates due to a timeout.

The object starts querying the server for data when the `readasync` function is called. Because all the data was sent before the `readasync` function call, no data will be stored in the input buffer and the data is lost.

When the TCP/IP object is in manual mode (the `ReadAsyncMode` property is configured to `manual`), data that is sent from the server to the computer is not automatically stored in the input buffer of the TCP/IP object. Data is not stored until `readasync` or one of the blocking read functions is called.

Manual mode should be used when a stream of data is being sent from your server and you only want to capture portions of the data.

Defining an Asynchronous Read Callback

You can configure a TCP/IP object to notify you when a terminator has been read using the `dispcallback` function.

```
t.ReadAsyncMode = 'continuous';
t.BytesAvailableFcn = 'dispcallback';
```

Note, the default value for the `BytesAvailableFcnMode` property indicates that the callback function defined by the `BytesAvailableFcn` property will be executed when the terminator has been read.

The callback function `dispcallback` displays event information for the specified event. Using the syntax `dispcallback(obj, event)`, it displays a message containing the type of event, the name of the object that caused the event to occur, and the time the event occurred.

```
callbackTime = datestr(datenum(event.Data.AbsTime));
fprintf(['A ' event.Type ' event occurred for ' obj.Name ' at '
    callbackTime '.\n']);
```

Using Callbacks During an Asynchronous Read

Once the terminator is read from the server and placed in the input buffer, `dispcallback` is executed and a message is posted to the MATLAB command window indicating that a `BytesAvailable` event occurred.

```
fprintf(t, 'Hello World 789')
t.BytesAvailable
ans =
    16
```

```
data = fscanf(t, '%c', 18)
data =
    Hello World 789
```

Note: If you need to stop an asynchronous read or write operation, you do not have to wait for the operation to complete. You can use the `stopasync` function to stop the asynchronous read or write.

Writing Data Asynchronously

You can perform an asynchronous write with the `fprintf` or `fwrite` functions by passing `'async'` as the last input argument.

In asynchronous mode, you can use callback properties and callback functions to perform tasks while data is being written. For example, configure the object to notify you when an asynchronous write operation completes.

```
t.OutputEmptyFcn = 'dispcallback';
fprintf(t, 'Hello World 123', 'async')
```

Note: If you need to stop an asynchronous read or write operation, you do not have to wait for the operation to complete. You can use the `stopasync` function to stop the asynchronous read or write.

Cleanup

If you are finished with the TCP/IP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(t);
delete(t);
clear t

echotcpip('off');
```

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 the MATLAB workspace, upload the data to the instrument, and then read back the waveform.

The instrument is a Sony/Tektronix AWG520 Arbitrary Waveform Generator (AWG). Its address is `sonytekawg.yourdomain.com` and its port is 4000. The AWG's host IP address is 192.168.1.10 and is user configurable in the instrument. The associated host name is given by your network administrator. The port number is fixed and is found in the instrument's documentation:

1   Create an instrument object — Create a TCP/IP object associated with the AWG.

```
t = tcpip('sonytekawg.yourdomain.com',4000);
```

2   Connect to the instrument — Before establishing a connection, the `OutputBufferSize` must be large enough to hold the data being written. In this example, 2577 bytes are written to the instrument. Therefore, the `OutputBufferSize` is set to `3000`.

```
t.OutputBufferSize = 3000)
```

You can now connect `t` to the instrument.

```
fopen(t)
```

3   Write and read data — Since the instrument's byte order is little-endian, configure the `ByteOrder` property to `littleEndian`.

```
t.ByteOrder = 'littleEndian'
```

Create the sine wave data.

```
x = (0:499).*8*pi/500;
data = sin(x);
marker = zeros(length(data),1);
marker(1) = 3;
```

Instruct the instrument to write the file `sin.wfm` with Waveform File format, a total length of 2544 bytes, and a combined data and marker length of 2500 bytes.

```
fprintf(t,'%s',['MMEMORY:DATA "sin.wfm",#42544MAGIC 1000' 13 10])
fprintf(t,'%s','#42500')
```

Write the sine wave to the instrument.

```
for (i = 1:length(data)),
 fwrite(t,data(i),'float32');
```

```
 fwrite(t,marker(i));
end
```

Instruct the instrument to use a clock frequency of 100 MS/s for the waveform.

```
fprintf(t,'%s',['CLOCK 1.0000000000e+008' 13 10 10])
```

Read the waveform stored in the function generator's hard drive. The waveform contains 2000 bytes plus markers, header, and clock information. To store this data, close the connection and configure the input buffer to hold 3000 bytes.

```
fclose(t)
t.InputBufferSize = 3000)
```

Reopen the connection to the instrument.

```
fopen(t)
```

Read the file `sin.wfm` from the function generator.

```
fprintf(t,'MMEMORY:DATA? "sin.wfm" ')
data = fread(t,t.BytesAvailable);
```

The next set of commands reads the same waveform as a `float32` array. To begin, write the waveform to the AWG.

```
fprintf(t,'MMEMORY:DATA? "sin.wfm" ')
```

Read the file header as ASCII characters.

```
header1 = fscanf(t)
header1 =
#42544MAGIC 1000
```

Read the next six bytes, which specify the length of data.

```
header2 = fscanf(t,'%s',6)
header2 =
#42500
```

Read the waveform using `float32` precision and read the markers using `uint8` precision. Note that one `float32` value consists of four bytes. Therefore, the following commands read 2500 bytes.

```
data = zeros(500,1);
```

```
marker = zeros(500,1);
for i = 1:500,
 data(i) = fread(t,1,'float32');
 marker(i) = fread(t,1,'uint8');
end
```

Read the remaining data, which consists of clock information and termination characters.

```
clock = fscanf(t);
cleanup = fread(t,2);
```

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

```
fclose(t)
delete(t)
clear t
```

Reading and Writing ASCII Data over UDP

This section explores ASCII read and write operations with a UDP object.

Functions and Properties

These functions are used when reading and writing text:

Function	Purpose
fprintf	Write text to the server.
fscanf	Read data from the server and format as text.

These properties are associated with reading and writing text:

Property	Purpose
ValuesReceived	Specifies the total number of values read from the server.
ValuesSent	Specifies the total number of values sent to the server.
InputBufferSize	Specifies the total number of bytes that can be queued in the input buffer at one time.
OutputBufferSize	Specifies the total number of bytes that can be queued in the output buffer at one time.

Property	Purpose
`Terminator`	Character used to terminate commands sent to the server.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the UDP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echoudp('on', 8000)
```

You need to create a UDP object. In this example, create a UDP object associated with the host 127.0.0.1 (your local machine), port 8000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
u = udp('127.0.0.1', 8000);
```

Before you can perform a read or write operation, you must connect the UDP object to the server with the `fopen` function.

```
fopen(u)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
u.Status
ans =
    open
```

Writing ASCII Data

You use the `fprintf` function to write ASCII data to the server. For example, write a string to the echoserver.

```
fprintf(u, 'Request Time');
```

By default, the `fprintf` function operates in a synchronous mode. This means that `fprintf` blocks the MATLAB command line until one of the following occurs:

- All the data is written
- A timeout occurs as specified by the `Timeout` property

By default the `fprintf` function writes ASCII data using the `%s\n` format. All occurrences of `\n` in the command being written to the server are replaced with the `Terminator` property value. When using the default format, `%s\n`, all commands written to the server will end with the `Terminator` character.

For the previous command, the linefeed (LF) is sent after `'Request Time'` is written to the server, thereby indicating the end of the command.

You can also specify the format of the command written by providing a third input argument to `fprintf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s.

For example, the data command previously shown can be written to the server using two calls to `fprintf`.

```
fprintf(u, '%s', 'Request');
fprintf(u, '%s'\n, 'Time');
```

The `Terminator` character indicates the end of the command and is sent after the last call to `fprintf`.

ASCII Write Properties

The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the server at once. By default, `OutputBufferSize` is `512`.

```
u.OutputBufferSize
ans =
    512
```

If the command specified in `fprintf` contains more than 512 bytes, an error is returned and no data is written to the server.

The `ValuesSent` property indicates the total number of values written to the server since the object was connected to the server.

```
u.ValuesSent
ans =
    26
```

Remove any data that was returned from the echoserver and captured by the UDP object.

```
flushinput(u);
```

Reading ASCII Data

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, a datagram with 13 bytes (12 ASCII bytes plus the LF terminator) is sent to the echoserver.

```
fprintf(u, 'Request Time');
```

The echo server will send back a datagram containing the same 13 bytes.

```
u.BytesAvailable
ans =
    13
```

You use the `fscanf` function to read ASCII data from the server.

```
data = fscanf(u)
data =
    Request Time
```

By default, the `fscanf` function reads data using the `'%c'` format and blocks the MATLAB command line until one of the following occurs:

- The terminator is received as specified by the `Terminator` property (if `DatagramTerminateMode` is `off`)
- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled
- The specified number of values is read (if `DatagramTerminateMode` is `off`)
- A datagram has been received (if `DatagramTerminateMode` is on)

You can also specify the format of the data read by providing a second input argument to `fscanf`. The accepted format conversion characters include: d, i, o, u, x, X, f, e, E, g, G, c, and s.

For example, the string `'0.80'` sent to the echoserver can be read into MATLAB as a double using the `%f` format string.

```
fprintf(u, '0.80');
data = fscanf(u, '%f')
data =
    0.8000
```

```
isnumeric(data)
ans =
     1
```

ASCII Read Properties

The `DatagramTerminateMode` property indicates whether a read operation should terminate when a datagram is received. By default `DatagramTerminateMode` is `on`, which means that a read operation terminates when a datagram is received. To read multiple datagrams at once, you can set `DatagramTerminateMode` to `off`. In this example, two datagrams are written. Note, only the second datagram sends the `Terminator` character.

```
fprintf(u, '%s', 'Request Time');
fprintf(u, '%s\n', 'Request Time');
```

Since `DatagramTerminateMode` is `off`, `fscanf` will read across datagram boundaries until the `Terminator` character is received.

```
u.DatagramTerminateMode = 'off';
data = fscanf(u)
data =
     Request TimeRequest Time
```

The `InputBufferSize` property specifies the maximum number of bytes you can read from the server. By default, `InputBufferSize` is `512`.

```
u.InputBufferSize
ans =
     512
```

The `ValuesReceived` property indicates the total number of values read from the server, including the terminator.

```
u.ValuesReceived
ans =
     43
```

Cleanup

If you are finished with the UDP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(u);
```

```
delete(u);
clear u

echoudp('off');
```

Reading and Writing Binary Data over UDP

This section explores binary read and write operations with a UDP object.

Functions and Properties

These functions are used when reading and writing binary data:

Function	Purpose
fread	Read binary data from the instrument or server.
fwrite	Write binary data to the instrument or server.

These properties are associated with reading and writing binary data:

Property	Purpose
ValuesReceived	Specifies the total number of values read from the instrument or server.
ValuesSent	Specifies the total number of values sent to the instrument or server.
InputBufferSize	Specifies the total number of bytes that can be queued in the input buffer at one time.
OutputBufferSize	Specifies the total number of bytes that can be queued in the output buffer at one time.
DatagramTerminate	Defines how fread and fscanf read operations terminate.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the UDP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echoudp('on', 8000)
```

You need to create a UDP object. In this example, create a UDP object associated with the host 127.0.0.1 (your local machine), port 8000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
u = udp('127.0.0.1', 8000);
```

You may need to configure the `OutputBufferSize` of the UDP object. The `OutputBufferSize` property specifies the maximum number of bytes that can be written to the server at once. By default, `OutputBufferSize` is `512`.

```
u.OutputBufferSize
ans =
    512
```

If the command specified in `fwrite` contains more than 512 bytes, an error is returned and no data is written to the server. In this example 1000 bytes will be written to the instrument. Therefore, the `OutputBufferSize` is increased to `1000`.

```
u.OutputBufferSize = 1000
u.OutputBufferSize
ans =
    1000
```

Before you can perform a read or write operation, you must connect the UDP object to the server with the `fopen` function.

```
fopen(u)
```

If the object was successfully connected, its `Status` property is automatically configured to `open`.

```
u.Status
ans =
    open
```

Writing Binary Data

You use the `fwrite` function to write binary data to the server or instrument.

By default, the `fwrite` function operates in a synchronous mode. This means that `fwrite` blocks the MATLAB command line until one of the following occurs:

• All the data is written

- A timeout occurs as specified by the `Timeout` property

By default the `fwrite` function writes binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fwrite`.

UDP sends and receives data in blocks that are called datagrams. Each time you write or read data with a UDP object, you are writing or reading a datagram. In the example below, a datagram with 1000 bytes, 4 bytes per integer number, will be sent to the echoserver.

```
fwrite(u, 1:250, 'int32');
```

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

The `ValuesSent` property indicates the total number of values written to the server since the object was connected to the server.

```
u.ValuesSent
ans =
    250
```

Configuring InputBufferSize

The `InputBufferSize` property specifies the maximum number of bytes that you can read from the server. By default, `InputBufferSize` is `512`.

```
u.InputBufferSize
ans =
    512
```

In the next example, 1000 bytes will be read from the server. Configure the `InputBufferSize` to hold 1000 bytes. Note, the `InputBufferSize` can be configured only when the object is not connected to the server or instrument.

```
fclose(u);
u.InputBufferSize = 1000);
u.InputBufferSize
ans =
    1000
```

Now that the property is configured correctly, you can reopen the connection to the server:

```
fopen(u);
```

Reading Binary Data

You use the `fread` function to read binary data from the server or instrument.

The `fread` function blocks the MATLAB command line until one of the following occurs:

- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled
- The specified number of values is read (if `DatagramTerminateMode` is `off`)
- A datagram has been received (if `DatagramTerminateMode` is `on`)

By default the `fread` function reads binary data using the `uchar` precision. However, other precisions can also be used. For a list of supported precisions, see the function reference page for `fread`.

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

You can read `int32` binary data. For example, read one datagram consisting of 250 integers from the instrument or server.

```
fwrite(u, 1:250, 'int32');

data = fread(u, 250, 'int32');
```

The `ValuesReceived` property indicates the total number of values read from the server.

```
u.ValuesReceived
ans =
    500
```

The `DatagramTerminateMode` property indicates whether a read operation should terminate when a datagram is received. By default `DatagramTerminateMode` is `on`, which means that a read operation terminates when a datagram is received. To read

multiple datagrams at once, you can set `DatagramTerminateMode` to `off`. In this example, two datagrams are written to the echoserver.

```
fwrite(u, 1:125, 'int32');
fwrite(u, 1:125, 'int32');
```

Because `DatagramTerminateMode` is `off`, `fread` will read across datagram boundaries until 250 integers have been received.

```
u.DatagramTerminateMode = 'off';
data = fread(u, 250, 'int32');
size(data)
ans =
    250
```

Cleanup

If you are finished with the UDP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(u);
delete(u);
clear u

echoudp('off');
```

Asynchronous Read and Write Operations over UDP

This section explores asynchronous read and write operations with a UDP object.

Functions and Properties

These functions are associated with reading and writing text asynchronously:

Function	Purpose
fprintf	Write text to a server.
readasync	Asynchronously read bytes from a server.
stopasync	Stop an asynchronous read or write operation.

These properties are associated with reading and writing text asynchronously:

Property	Purpose
BytesAvailable	Indicates the number of bytes available in the input buffer.

Property	Purpose
TransferStatus	Indicates what type of asynchronous operation is in progress.
ReadAsyncMode	Indicates whether data is read continuously in the background or whether you must call the readasync function to read data asynchronously.

Additionally, you can use all the callback properties during asynchronous read and write operations.

Synchronous Versus Asynchronous Operations

The object can operate in synchronous mode or in asynchronous mode. When the object is operating synchronously, the read and write routines block the MATLAB command line until the operation has completed or a timeout occurs. When the object is operating asynchronously, the read and write routines return control immediately to the MATLAB command line.

Additionally, you can use callback properties and callback functions to perform tasks as data is being written or read. For example, you can create a callback function that notifies you when the read or write operation has finished.

Configuring and Connecting to the Server

For this example, we will use an echo server that is provided with the toolbox. The echo server allows you to experiment with the basic functionality of the UDP objects without connecting to an actual device. An echo server is a service that returns to the sender's address and port, the same bytes it receives from the sender.

```
echoudp('on', 8000);
```

You need to create a UDP object. In this example, create a UDP object associated with the host 127.0.0.1 (your local machine), port 8000. In general, the host name or address and the host port will be defined by the device and your network configuration.

```
u = udp('127.0.0.1', 8000);
```

Before you can perform a read or write operation, you must connect the UDP object to the server with the fopen function.

```
fopen(u)
```

If the object was successfully connected, its Status property is automatically configured to open.

```
u.Status
ans =
    open
```

Reading Data Asynchronously

You can read data asynchronously with the UDP object in one of these two ways:

- Continuously, by setting `ReadAsyncMode` to `continuous`. In this mode, data is automatically stored in the input buffer as it becomes available from the server.
- Manually, by setting `ReadAsyncMode` to `manual`. In this mode, you must call the `readasync` function to store data in the input buffer.

The `fscanf`, `fread`, `fgetl` and `fgets` functions are used to bring the data from the input buffer into MATLAB. These functions operate synchronously.

Reading Data Asynchronously Using Continuous ReadAsyncMode

To read data continuously:

```
u.ReadAsyncMode = 'continuous';
```

To send a string to the echoserver:

```
fprintf(u, 'Hello net.');
```

Because the `ReadAsyncMode` property is set to `continuous`, the object is continuously asking the server if any data is available. The echoserver sends data as soon as it receives data. The data is then read from the server and is stored in the object's input buffer.

```
u.BytesAvailable
ans =
    11
```

You can bring the data from the object's input buffer into the MATLAB workspace with `fscanf`.

```
mystring = fscanf(u)
mystring =
    Hello net.
```

Reading Data Asynchronously Using Manual ReadAsyncMode

You can also read data manually.

```
u.ReadAsyncMode = manual;
```

Now, send a string to the echoserver.

```
fprintf(u, 'Hello net.');
```

Once the last `fprintf` function completes, the server begins sending data. However, because `ReadAsyncMode` is set to `manual`, the object is not reading the data being sent from the server. Therefore no data is being read and placed in the input buffer.

```
u.BytesAvailable
ans =
    0
```

The `readasync` function can asynchronously read the data from the server. The `readasync` function returns control to the MATLAB command line immediately.

The `readasync` function takes two input arguments. The first argument is the server object and the second argument is the `size`, the amount of data to be read from the server.

The `readasync` function without a `size` specified assumes `size` is given by the difference between the `InputBufferSize` property value and the `BytesAvailable` property value. The asynchronous read terminates when:

- The terminator is read as specified by the `Terminator` property
- The specified number of bytes have been read
- A timeout occurs as specified by the `Timeout` property
- The input buffer is filled

An error event will be generated if `readasync` terminates due to a timeout.

The object starts querying the server for data when the `readasync` function is called. Because all the data was sent before the `readasync` function call, no data will be stored in the input buffer and the data is lost.

When the UDP object is in manual mode (the `ReadAsyncMode` property is configured to `manual`), data that is sent from the server to the computer is not automatically stored in the input buffer of the UDP object. Data is not stored until `readasync` or one of the blocking read functions is called.

Manual mode should be used when a stream of data is being sent from your server and you only want to capture portions of the data.

Defining an Asynchronous Read Callback

You can configure a UDP object to notify you when a terminator has been read using the dispcallback function.

```
u.ReadAsyncMode = 'continuous';
u.BytesAvailableFcn = 'dispcallback';
```

Note, the default value for the BytesAvailableFcnMode property indicates that the callback function defined by the BytesAvailableFcn property will be executed when the terminator has been read.

```
u.BytesAvailableFcnMode
ans =
    terminator
```

The callback function dispcallback displays event information for the specified event. Using the syntax dispcallback(obj, event), it displays a message containing the type of event, the name of the object that caused the event to occur, and the time the event occurred.

```
callbackTime = datestr(datenum(event.Data.AbsTime));
fprintf(['A ' event.Type ' event occurred for ' obj.Name ' at '
    callbackTime '.\n']);
```

Using Callbacks During an Asynchronous Read

Once the terminator is read from the server and placed in the input buffer, dispcallback is executed and a message is posted to the MATLAB command window indicating that a BytesAvailable event occurred.

```
fprintf(u, 'Hello net.')
u.BytesAvailable
ans =
    11

data = fscanf(u)
data =
    Hello net.
```

If you need to stop an asynchronous read or write operation, you do not have to wait for the operation to complete. You can use the stopasync function to stop the asynchronous read or write.

```
stopasync(u);
```

The data that has been read from the server remains in the input buffer. You can use one of the synchronous read functions to bring this data into the MATLAB workspace. However, because this data represents the wrong data, the `flushinput` function is called to remove all data from the input buffer.

```
flushinput(u);
```

Writing Data Asynchronously

You can perform an asynchronous write with the `fprintf` or `fwrite` functions by passing `'async'` as the last input argument.

Configure the object to notify you when an asynchronous write operation completes.

```
u.OutputEmptyFcn = 'dispcallback';
fprintf(u, 'Hello net.', 'async')
```

UDP sends and receives data in blocks that are called datagrams. Each time you write or read data with a UDP object, you are writing or reading a datagram. In the example below, a datagram with 11 bytes (10 ASCII bytes plus the LF terminator) will be sent to the echoserver. Then the echoserver will send back a datagram containing the same 11 bytes.

Configure the object to notify you when a datagram has been received.

```
u.DatagramReceivedFcn = 'dispcallback';
fprintf(u, 'Hello net.', 'async')
```

Note: If you need to stop an asynchronous read or write operation, you do not have to wait for the operation to complete. You can use the `stopasync` function to stop the asynchronous read or write.

Cleanup

If you are finished with the UDP object, disconnect it from the server, remove it from memory, and remove it from the workspace. If you are using the echo server, turn it off.

```
fclose(u);
delete(u);
```

```
clear u

echoudp('off');
```

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.

```
u.DatagramTerminateMode = 'off'
data = fscanf(u)
data =
Request TimeRequest 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.

```
fclose(u)
delete(u)
clear u
```

Events and Callbacks

In this section...
"Event Types and Callback Properties" on page 7-48
"Responding To Event Information" on page 7-49
"Using Events and Callbacks" on page 7-50

Event Types and Callback Properties

The event types and associated callback properties supported by TCP/IP and UDP objects are listed below.

TCP/IP and UDP Event Types and Callback Properties

Event Type	Associated Properties
Bytes available	`BytesAvailableFcn`
	`BytesAvailableFcnCount`
	`BytesAvailableFcnMode`
Datagram received	`DatagramReceivedFcn` (UDP objects only)
Error	`ErrorFcn`
Output empty	`OutputEmptyFcn`
Timer	`TimerFcn`
	`TimerPeriod`

The datagram-received event is described below. For a description of the other event types, refer to "Event Types and Callback Properties" on page 4-30.

Note: You cannot use ASCII values larger than 127 characters with `fgetl`, `fgets`, or `BytesAvailableFnc`. The functions are limited to 127 binary characters.

Datagram-Received Event

A datagram-received event is generated immediately after a complete datagram is received in the input buffer.

This event executes the callback function specified for the `DatagramReceivedFcn` property. It can be generated for both synchronous and asynchronous read operations.

Responding To Event Information

You can respond to event information in a callback function or in a record file. Event information stored in a callback function uses two fields: `Type` and `Data`. The `Type` field contains the event type, while the `Data` field contains event-specific information. 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 16-6 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

Event Type	Field	Field Value
Bytes available	`Type`	`BytesAvailable`
	`Data.AbsTime`	day-month-year hour:minute:second
Datagram received	`Type`	`DatagramReceived`
	`Data.AbsTime`	day-month-year hour:minute:second
	`Data.DatagramAddress`	IP address string
	`Data.DatagramLength`	Number of bytes received as double
	`Data.DatagramPort`	Port number of sender as double
Error	`Type`	`Error`
	`Data.AbsTime`	day-month-year hour:minute:second
	`Data.Message`	An error string
Output empty	`Type`	`OutputEmpty`
	`Data.AbsTime`	day-month-year hour:minute:second

Event Type	Field	Field Value
Timer	`Type`	`Timer`
	`Data.AbsTime`	day-month-year hour:minute:second

The `Data` field values are described below.

AbsTime Field

`AbsTime` is defined for all events, and indicates the absolute time the event occurred. The absolute time is returned using the MATLAB Command window `clock` format. day-month-year hour:minute:second

DatagramAddress Field

`DatagramAddress` is the IP address of the datagram sender.

DatagramLength Field

`DatagramLength` is the length of the datagram in bytes.

DatagramPort Field

`DatagramPort` is the sender's port number from which the datagram originated.

Message Field

`Message` is used by the error event to store the descriptive message that is generated when an error occurs.

Using Events and Callbacks

This example extends "Communicating Between Two Hosts" on page 7-11 to include a datagram received callback. The callback function is `instrcallback`, which displays information to the command line indicating that a datagram has been received.

The following command configures the callback for the UDP object `u2`.

```
u2.DatagramReceivedFcn = @instrcallback;
```

When a datagram is received, the following message is displayed.

```
DatagramReceived event occurred at 10:26:20 for the object:
UDP-doetom.dhpc.
25 bytes were received from address 192.168.1.12, port 8844.
```

Note: You cannot use ASCII values larger than 127 characters with `fgetl`, `fgets`, or `BytesAvailableFnc`. The functions are limited to 127 binary characters.

Using TCP/IP Server Sockets

In this section...
"About Server Sockets" on page 7-52
"Example" on page 7-52

About Server Sockets

Support for Server Sockets is available, using the `NetworkRole` property on the TCP/IP interface. This support is for a single remote connection. You can use this connection to communicate between a client and MATLAB, or between two instances of MATLAB.

For example, you might collect data such as a waveform into one instance of MATLAB, and then want to transfer it to another instance of MATLAB.

Note: The use of the server socket on either the client or server side should be done in accordance with the license agreement as it relates to your particular license option and activation type. If you have questions, you should consult with the administrator for your license or your legal department.

This is intended for use behind a firewall on a private network.

Note that while a server socket is waiting for a connection after calling `fopen`, the MATLAB processing thread is blocked. To stop `fopen` or to stop listening for connections, and restore the use of MATLAB, type **Ctrl+C** at the MATLAB command line.

Example

To use this feature it is necessary to set the `NetworkRole` property in the `tcpip` interface. It uses two values, `client` and `server`, to establish a connection as the client or the server. The server sockets feature supports binary and ASCII transfers.

The following example shows how to connect two MATLAB sessions on the same computer, showing the example code for each session. To use two different computers, replace `'localhost'` with the IP address of the server in the code for Session 2. Using '0.0.0.0' as the IP address means that the server will accept the first machine that tries

to connect. To restrict the connections that will be accepted, replace `'0.0.0.0'` with the address of the client in the code for Session 1.

Session 1: MATLAB Server

Accept a connection from any machine on port 30000.

```
t = tcpip('0.0.0.0', 30000, 'NetworkRole', 'server');
```

Open a connection. This will not return until a connection is received.

```
fopen(t);
```

Read the waveform and confirm it visually by plotting it.

```
data = fread(t, t.BytesAvailable);
plot(data);
```

Session 2: MATLAB Client

This code is running on a second copy of MATLAB.

Create a waveform and visualize it.

```
data = sin(1:64);
plot(data);
```

Create a client interface and open it.

```
t = tcpip('localhost', 30000, 'NetworkRole', 'client');
fopen(t)
```

Write the waveform to the server session.

```
fwrite(t, data)
```

8

Controlling Instruments Using Bluetooth

Bluetooth Interface Overview

In this section...
"Bluetooth Communication" on page 8-2
"Supported Platforms for Bluetooth" on page 8-2

Bluetooth Communication

The Instrument Control Toolbox Bluetooth interface lets you connect to devices over the Bluetooth interface and to transmit and receive ASCII and binary data. Instrument Control Toolbox supports the Bluetooth Serial Port Profile (SPP). You can identify any SPP Bluetooth device and establish a two-way connection with that device.

Bluetooth is an open wireless technology standard for exchanging data over short distances using short wavelength radio transmissions from fixed and mobile devices using a packet-based protocol. Bluetooth provides a secure way to connect and exchange information between devices such as Lego Mindstorm NXT robots, USB Bluetooth adaptors (dongles), wireless sensors, mobile phones, faxes, laptops, computers, printers, GPS receivers, etc.

Specifications about the Bluetooth standard are at the web site of the Bluetooth Special Interest Group:

```
https://www.bluetooth.org/Technical/Specifications/adopted.htm
```

Supported Platforms for Bluetooth

The Bluetooth interface is supported on these platforms:

- Mac OS 10.7 and earlier versions only 64-bit
- Microsoft Windows 32-bit
- Microsoft Windows 64-bit

Configuring Bluetooth Communication

In this section...
"Discovering Your Device" on page 8-3
"Viewing Bluetooth Device Properties" on page 8-5

Discovering Your Device

Instrument Control Toolbox can communicate with Bluetooth devices via an adaptor. In this example, a USB Bluetooth adaptor is plugged into the computer. It can identify Bluetooth devices within range when queried. In order to communicate with instruments, you need to perform a pairing in the adaptor software. Note that some devices, such as many laptop computers, do not need to use an adaptor since they have one built in.

The following shows the software interface of an adaptor where two of the devices in range have been paired – a smart phone with Bluetooth enabled, and a Lego Mindstorm NXT robot. As you can see, the "friendly name" or display name of the smart phone is simply `iPhone` and the name of the NXT robot is `C3PO`. In the Instrument Control Toolbox this friendly name is the Bluetooth `RemoteName` property.
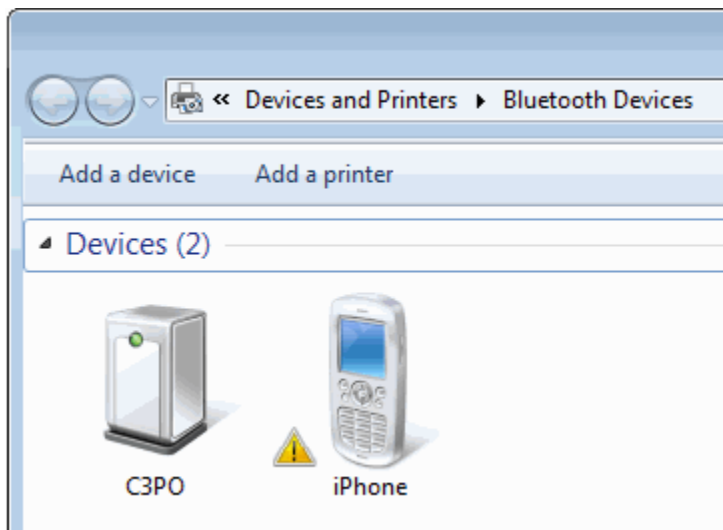
To see the devices in the Instrument Control Toolbox, use the `instrhwinfo` function on the Bluetooth interface, called `Bluetooth`.

```
>> instrhwinfo('Bluetooth')

ans =

         RemoteNames: {5x1 cell}
           RemoteIDs: {5x1 cell}
     BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
      JarFileVersion: 'Version 3.0.0'

>> b = instrhwinfo('Bluetooth')

b =

         RemoteNames: {5x1 cell}
           RemoteIDs: {5x1 cell}
     BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
      JarFileVersion: 'Version 3.0.0'
```

`instrhwinfo` returned a cell array of five Bluetooth devices that are in the range of the adaptor on the computer running Instrument Control Toolbox. Then indexing into the `RemoteNames` property shows the five devices. You can see that `iPhone` and `C3PO` are shown in the list.

```
>> b.RemoteNames

ans =

    'iPhone'
    'mprocopi-maci'
    'C3PO'
    ''
    'Emulator'
```

Notice that one of the other devices shows an empty string for `RemoteName`. That means that device does not have a friendly name associated with it. To communicate with that device, you need to use the `RemoteID` property.

```
>> b.RemoteIDs

ans =

    'E0F847D773E4'
    '001F5BDC815E'
    '0016530FD63D'
    '0021BA74F3DD'
    '00000000000'
```

The `RemoteIDs` are shown in the same order as the `RemoteNames`, so the fourth ID in the list, `'0021BA74F3DD'`, could be used for the device that shows no `RemoteName`. You can use either `RemoteName` or `RemoteID` to communicate with a device.

Examples of communicating with a device are in "Transmitting Data Over the Bluetooth Interface" on page 8-10.

Viewing Bluetooth Device Properties

This example looks at the NXT robot discovered in the previous section. Using the `instrhwinfo` function on the specific device using `RemoteName` shows this:

```
>> instrhwinfo('Bluetooth','C3PO')

ans =

              RemoteName: 'C3PO'
                DeviceID: '0016530FD63D'
    ObjectConstructorName: {'Bluetooth('C3PO', 1);'}
                Channels: {'1'}
```

If you use the `instrhwinfo` function on the specific device using the `RemoteID`, it shows the following:

```
>> instrhwinfo('Bluetooth','0016530FD63D')

ans =

                RemoteName: 'C3PO'
                  DeviceID: '0016530FD63D'
     ObjectConstructorName: {'Bluetooth('btspp://0016530FD63D', 1);'}
                  Channels: {'1'}
```

In the case using the `RemoteID`, you can see that the `ObjectConstructorName` is actually the device's Uniform Resource Identifier (URI).

Whether you use the `RemoteName` or the `RemoteID` to see the device's properties, you can see that the device has only one channel. Create a Bluetooth object `bt` using the `RemoteName` and `Channel`. Then display the state of that object using the `disp` function.

```
>> bt = Bluetooth('C3PO', 1);
>> disp(bt);

   Bluetooth Object : Bluetooth-C3PO:1

   Communication Settings
        RemoteName:          C3PO
        RemoteID:            0016530FD63D
        Channel:             1
        Terminator:          'LF'

   Communication State
        Status:              closed
        RecordStatus:        off

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

The status is `closed` because you have not yet opened the connection to the object.

Use the `get` function to see the device properties.

```
>> get(bt)
    ByteOrder = littleEndian
    BytesAvailable = 0
    BytesAvailableFcn =
    BytesAvailableFcnCount = 48
    BytesAvailableFcnMode = terminator
    BytesToOutput = 0
    ErrorFcn =
    InputBufferSize = 512
    Name = Bluetooth-C3PO:1
    ObjectVisibility = on
    OutputBufferSize = 512
    OutputEmptyFcn =
    RecordDetail = compact
    RecordMode = overwrite
    RecordName = record.txt
    RecordStatus = off
    Status = closed
    Tag =
    Timeout = 10
    TimerFcn =
    TimerPeriod = 1
    TransferStatus = idle
    Type = bluetooth
    UserData = []
    ValuesReceived = 0
    ValuesSent = 0

    BLUETOOTH specific properties:
    Channel = 1
    Profile = SPP
    ReadAsyncMode = continuous
    RemoteID = 0016530FD63D
    RemoteName = C3PO
    Terminator = LF
```

The `BLUETOOTH specific properties` section shows properties that are specific to the Bluetooth interface. You can see it is using channel 1. The `profile` is `SPP`, which is the Serial Port Profile – that is the Bluetooth profile that Instrument Control Toolbox supports.

The `RemoteName` and `RemoteID` properties are the names that are used to communicate with the device, as shown previously.

The `ReadAsyncMode` and `Terminator` properties are the same as the Serial Port properties of the same name. For details, see the properties documentation.

Transmitting Data Over the Bluetooth Interface

You can read and write both text data (ASCII based) and binary data. For text data, use the `fscanf` and `fprintf` functions. For binary data, use the `fread` and `fwrite` functions.

This example uses the LEGO Mindstorm NXT robot with a `RemoteName` of `C3PO` that you discovered in "Viewing Bluetooth Device Properties" on page 8-5. See that section for more details on device discovery and viewing properties.

To communicate with the NXT device:

1 Determine what Bluetooth devices are accessible from your computer.

```
instrhwinfo('Bluetooth')
```

2 View the device list using the `RemoteNames` property.

```
ans.RemoteNames
```

3 In this case, `C3PO` is the remote name of the NXT robot and is shown in the output. Display the information about this device using the `Bluetooth` interface and the `RemoteName` property.

```
instrhwinfo('Bluetooth','C3PO')
```

The Instrument Control Toolbox displays the device information.

```
>> instrhwinfo('Bluetooth','C3PO')

ans =

                RemoteName: 'C3PO'
                  DeviceID: '0016530FD63D'
     ObjectConstructorName: {'Bluetooth('C3PO', 1);'}
                  Channels: {'1'}
```

4 Create a Bluetooth object called `bt` using channel 1 of the NXT device.

```
bt = Bluetooth('C3PO', 1);
```

5 Connect to the device.

```
fopen(bt)
```

6 Send a message to the remote device using the `fwrite` function.

In this example, specific characters are sent to the device that this particular device (the NXT robot `C3PO`) understands. You can write to the device, then query the object, as shown here, to see that the values were sent.

```
>> fwrite(bt, uint8([2,0,1,155]));
>> bt

   Bluetooth Object : Bluetooth-C3PO:1

   Communication Settings
      RemoteName:          C3PO
      RemoteID:            0016530FD63D
      Channel:             1
      Terminator:          'LF'

   Communication State
      Status:              open
      RecordStatus:        off

   Read/Write State
      TransferStatus:      idle
      BytesAvailable:      35
      ValuesReceived:      0
      ValuesSent:          4
```

7 Read data from the remote device using the `fread` function.

You can see that `ValuesSent` is 4, which are the four characters sent in `fwrite` (`2,0,1,155`). This also shows that `35` bytes are available. So you can then use the `fread` function and give it 35 bytes to read the characters from the remote device.

```
>> name = fread(bt,35);
```

```
>> char(name(6:10)')

ans =

C3PO
```

The device returns the characters shown here. The returned characters are `C3PO`, which is the `RemoteName` of the device. That was a reply to the instructions that were sent to it. See the documentation for your device for this type of device-specific communication information.

8 Clean up by deleting and clearing the object.

```
fclose(bt);
clear(bt);
```

Note: This example uses the `fread` and `fwrite` functions. To read and write text-based data, use the `fscanf` and `fprintf` functions.

Note: You can do asynchronous reading and writing of data using the Bluetooth interface. This is similar to the same operations using the Instrument Control Toolbox Serial interface. For more information, see "Asynchronous Write and Read Operations" on page 6-18.

Other Functionality

The following functions can be used with the Bluetooth object.

Function	Purpose
binblockwrite	Write binblock data to instrument
fgetl	Read line of text from instrument and discard terminator
flushinput	Remove data from input buffer
fopen	Connect interface object to instrument
fread	Read binary data from instrument
fwrite	Write binary data to instrument
methods	Class method names and descriptions
readasync	Read data asynchronously from instrument
scanstr	Read data from instrument, format as text, and parse
binblockread	Read binblock data from instrument
fclose	Disconnect interface object from instrument
fgets	Read line of text from instrument and include terminator
flushoutput	Remove data from output buffer
fprintf	Write text to instrument
fscanf	Read data from instrument, and format as text
query	Write text to instrument, and read data from instrument
record	Record data and event information to file
stopasync	Stop asynchronous read and write operations

For more information about these functions, see the functions documentation.

Using Bluetooth Interface in Test & Measurement Tool

The Bluetooth interface is supported for use in the Test & Measurement Tool. The same functionality is available there as in the core toolbox, as described in "Bluetooth Interface Overview" on page 8-2.

To use the Bluetooth support in the Test & Measurement Tool, select the **Bluetooth** node in the instrument tree and right-click **Scan for bluetooth devices**. For help on using it in the Test & Measurement Tool, see the Help within the tool by selecting the **Bluetooth** node in the tree and reading the **Help** panel.

Note: When using the Bluetooth support in the Test & Measurement Tool, please note that you may need to restart your device after you have done the scan. For any Lego Mindstorm robot to be identified correctly, it has to be restarted after scanning. You may also have to restart other Bluetooth devices after the scan as well.

Troubleshooting

If you are having trouble using the Bluetooth interface in the Test & Measurement Tool, try these steps.

- Check that the Bluetooth device supports the Serial Port Profile (SPP). We do not support other Bluetooth profiles such as File Transfer Profile (FTP).
- Make sure that the Bluetooth service on the device is turned on.
- Make sure that the Bluetooth device is paired with your computer.
- If you are using a Lego Mindstorm NXT brick, note that the NXT brick has to be restarted after scanning for Bluetooth devices from the Test & Measurement Tool.
- If you still cannot connect to the Bluetooth device, try unplugging and replugging the Bluetooth adaptor.

Note: For further information on using the Bluetooth interface, see "Bluetooth Interface Usage Guidelines" on page 8-16.

Using Events and Callbacks with Bluetooth

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 MATLAB 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 callback function as the value for the associated callback property.

The Bluetooth event types and associated callback properties are described below.

Event Type	Associated Property Name

Bytes-available	`BytesAvailableFcn`
	
	 `BytesAvailableFcnCount`
	
	 `BytesAvailableFcnMode`

Error	`ErrorFcn`
Output-empty	`OutputEmptyFcn`

Timer	`TimerFcn`
	
	 `TimerPeriod`

These are the same callbacks that are commonly used by other interfaces in the Instrument Control Toolbox.

Bluetooth Interface Usage Guidelines

These guidelines may be relevant to your use of this feature.

On Windows 7 64-bit platforms, you can use only one Bluetooth adaptor at a time. If you connect another adaptor, it will fail with a "Device Driver Installation Failed" error.

Some adaptors support multiple devices:

- The Bluetooth adaptor that comes with the LEGO Mindstorm kit (Abe – Model: UB22S) supports connection to only one Bluetooth device at a time.
- IO Gear – Models GBU421 and GBU311 support communication with multiple Bluetooth devices.
- Targus – Model ACB10US supports communication with multiple Bluetooth devices.
- Motorola – Model SYN1244B supports communication with multiple Bluetooth devices.
- D-Link – Model DBT-120 supports communication with multiple Bluetooth devices.

If a Bluetooth adaptor is removed and a different one plugged in, all Bluetooth devices have to be paired again with your PC. If the same adaptor is removed and plugged back in, then you do not need to pair the devices again. If another adaptor of the same vendor is plugged in, then the devices which had been cached when that adaptor was used are seen in the cache.

If a Bluetooth device is already cached, but it is OFF when MATLAB is started, and if `instrhwinfo` is called on this device, then `ObjectConstructorName` and `Channel` are returned as a null string. If a Bluetooth device is already cached and is ON when MATLAB is started, and it is later switched OFF, if `instrhwinfo` is called on this device, then `ObjectConstructorName` and `Channel` return the correct values.

If you create a Bluetooth object for any Bluetooth device and the connection is open, and then the device goes out of range, then the status of the object would still be `open`. When the device comes into range again, you need to `fclose` the object and `fopen` it again for communication to continue.

If you create a Bluetooth object, for a Lego Mindstorm NXT robot for example, and the connection is open, and then the batteries of robot run out, then the status of the object would still be `open`. If you then replace the batteries, you need to `fclose` the object and `fopen` it again for communication to continue.

When using the Bluetooth support in the Test & Measurement Tool, please note that you may need to restart your device after you have done the scan. For any Lego Mindstorm robot to be identified correctly, it has to be restarted after scanning. You may also have to restart other Bluetooth devices after the scan as well.

9

Controlling Instruments Using I2C

I2C Interface Overview

In this section...
"I2C Communication" on page 9-2
"Supported Platforms for I2C" on page 9-2

I2C Communication

I2C, or Inter-Integrated Circuit, is a chip-to-chip interface supporting two-wire communication. Instrument Control Toolbox I2C support lets you open connections with individual chips and to read and write over the connections to individual chips.

The Instrument Control Toolbox I2C interface lets you do chip to chip communication using an Aardvark or NI-845x host adaptor. Some applications of this interface include communication with SPD EEPROM and NVRAM chips, communication with SMBus devices, controlling accelerometers, accessing low-speed DACs and ADCs, changing settings on color monitors using the display data channel, changing sound volume in intelligent speakers, reading hardware monitors and diagnostic sensors, visualizing data sent from an I2C sensor, and turning on or off the power supply of system components.

The primary use cases involve the `fread` and `fwrite` functions. To identify I2C devices in the Instrument Control Toolbox, use the `instrhwinfo` function on the I2C interface, called `i2c`.

Supported Platforms for I2C

You need to have either a Total Phase Aardvark host adaptor or a NI-845x adaptor board installed to use the `i2c` interface. The following sections contain the supported platforms for each option.

Using Aardvark

The I2C interface is supported on these platforms when used with the Aardvark host adaptor:

- Linux – The software works with Red Hat Enterprise Linux 4 and 5 with kernel 2.6. It may also be successful with SuSE and Ubuntu distributions.
- Mac OS X 64-bit – The software is supported on Intel versions of Mac OS X 10.5 Leopard and 10.6 Snow Leopard.
- Microsoft Windows 32-bit – The software is supported on Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), and Windows 7 (32-bit and 64-bit).
- Microsoft Windows 64-bit – The software is supported on Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), and Windows 7 (32-bit and 64-bit).

Using NI-845x

The I2C interface is supported on these platforms when used with the NI-845x host adaptor:

- Microsoft Windows 32-bit – The software is supported on Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), and Windows 7 (32-bit and 64-bit).
- Microsoft Windows 64-bit – The software is supported on Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), and Windows 7 (32-bit and 64-bit).

Configuring I2C Communication

You need to have either a Total Phase Aardvark host adaptor or a NI-845x adaptor board installed to use the `i2c` interface. The following sections describe configuration for each option.

Configuring Aardvark

To use the I2C interface with the Aardvark adaptor, you must download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the Aardvark driver installed, see "Install the Total Phase Aardvark I2C/SPI Support Package" on page 14-9 to install it via the Support Package Installer.

The aardvark.dll file that comes with the Total Phase Aardvark adaptor board must be available in one of the following locations for use on Windows platforms.

- The location where MATLAB was started from (Bin folder).
- The MATLAB current folder (PWD).
- The Windows folder `C:\winnt` or `C:\windows`.
- The folders listed in the PATH environment variable.

Ensure that the Aardvark adaptor is installed properly.

```
>> instrhwinfo('i2c')

ans =

    InstalledAdaptors: 'Aardvark'
       JarFileVersion: 'Version 3.0.0'
```

Look at the adaptor properties.

```
instrhwinfo('i2c', 'Aardvark')

ans =

          AdaptorDllName: [1x127 char]
       AdaptorDllVersion: 'Version 3.0.0'
             AdaptorName: 'aardvark'
       InstalledBoardIds: 0
   ObjectConstructorName: 'i2c('aardvark', BoardIndex, RemoteAddress);'
           VendorDllName: 'aardvark.dll'
 VendorDriverDescription: 'Total Phase I2C Driver'
```

You can create an I2C object using the `i2c` function. The example in the next section uses an I2C object called eeprom that communicates to an EEPROM chip.

```
eeprom = i2c('aardvark',0,hex2dec('50'));
```

You can then display the object properties.

```
>> disp(eeprom)

   I2C Object : I2C-0-50h

   Communication Settings
      BoardIndex         0
      BoardSerial        2237482577
      BitRate:           100
      RemoteAddress:     50h
      Vendor:            aardvark


   Communication State
      Status:            open
      RecordStatus:      off

   Read/Write State
      TransferStatus:    idle
      BytesAvailable:    0
      ValuesReceived:    16
      ValuesSent:        15
```

You can see that the communication settings properties reflect what was used to create the object – `BoardIndex` of `0` and `RemoteAddress` of `50h`. For information about other properties, see "Using Properties on an I2C Object" on page 9-14.

Configuring NI-845x

To use the I2C interface with the NI-845x adaptor, you must download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the NI-845x driver installed, see "Install the NI-845x I2C/SPI Interface Support Package" on page 14-8 to install it via the Support Package Installer.

Ensure that the NI-845x adaptor is installed properly.

```
>> instrhwinfo('i2c')

ans =

    InstalledAdaptors: {'Aardvark'  'NI845x'}
        JarFileVersion: 'Version 3.4'
```

Look at the NI-845x adaptor properties.

```
>> instrhwinfo('i2c', 'NI845x')

ans =

              AdaptorDllName: [1x120 char]
           AdaptorDllVersion: 'Version 3.4'
                 AdaptorName: 'NI845x'
               BoardIdsInUse: [1x0 double]
           InstalledBoardIDs: 0
         DetectedBoardSerials: {'0180D47A (BoardIndex: 0)'}
        ObjectConstructorName: 'i2c('NI845x', BoardIndex, RemoteAddress);'
              VendorDllName: 'Ni845x.dll'
      VendorDriverDescription: 'National Instruments NI USB 845x Driver'
```

You can create an I2C object using the `i2c` function.

```
i2cobj = i2c('NI845x', 0, '10h');
```

You can then display the object properties.

```
>> disp(i2cobj)

    I2C Object : I2C-0-10h

    Communication Settings
        BoardIndex          0
        BoardSerial         0180D47A
        BitRate:            100 kHz
        RemoteAddress:      10h
        Vendor:             NI845x


    Communication State
        Status:             closed
        RecordStatus:       off

    Read/Write State
        TransferStatus:     idle
```

You can see that the communication settings properties reflect what was used to create the object – BoardIndex of 0 and RemoteAddress of 10h. For information about other properties, see "Using Properties on an I2C Object" on page 9-14.

Transmitting Data Over the I2C Interface

The typical workflow involves adaptor discovery, connection, communication, and cleanup. Discovery can be done only at the adaptor level. You need to have either a Total Phase Aardvark host adaptor or a NI-845x adaptor board installed to use the `i2c` interface.

Aardvark Example

This example shows how to communicate with an EEPROM chip on a circuit board, with an address of 50 hex and a board index of 0, using the Aardvark adaptor.

To communicate with an EEPROM chip:

1 Ensure that the Aardvark adaptor is installed so that you can use the `i2c` interface.

```
instrhwinfo('i2c')

>> instrhwinfo('i2c')


ans =


    InstalledAdaptors: 'Aardvark'
        JarFileVersion: 'Version 3.0.0'
```

2 Look at the adaptor properties.

```
instrhwinfo('i2c', 'Aardvark')

ans =

           AdaptorDllName: [1x127 char]
        AdaptorDllVersion: 'Version 3.0.0'
              AdaptorName: 'aardvark'
         InstalledBoardIds: 0
    ObjectConstructorName: 'i2c('aardvark', BoardIndex, RemoteAddress);'
            VendorDllName: 'aardvark.dll'
 VendorDriverDescription: 'Total Phase I2C Driver'
```

Make sure that you have the Aardvark software driver installed and that the aardvark.dll is on your MATLAB path. For details, see "I2C Interface Usage Requirements and Guidelines" on page 9-17.

3 Create the I2C object called `eeprom`, using these properties:

```
% Vendor = aardvark
% BoardIndex = 0
% RemoteAddress = 50h

eeprom = i2c('aardvark',0,'50h');
```

You must provide these three parameters to create the object. Read the documentation of the chip in order to know what the remote address is.

Tip You can also see what the remote address of the chip is by scanning for instruments in the Test & Measurement Tool. In the tool, right-click the **I2C** node and select **Scan for I2C adaptors**. Any chips found by the scan is listed in the hardware tree. The listing includes the remote address of the chip.

---

4 Connect to the chip.

```
fopen(eeprom);
```

5 Write `'Hello World!'` to the EEPROM. Data is written page-by-page in I2C. Each page contains eight bytes. The page address needs to be mentioned before every byte of data written.

The first byte of the string `'Hello World!'` is `'Hello Wo'`. Its page address is 0.

```
fwrite(eeprom,[0 'Hello Wo']);
```

The second byte of the string `'Hello World!'` is `'rld!'`. Its page address is 8.

```
fwrite(eeprom,[8 'rld!']);
```

6 Read data back from the chip using the `fread` function.

A zero needs to be written to the `i2c` object, to start reading from the first byte of first page.

```
fwrite(eeprom,0);

char(fread(eeprom,16))'
```

The chip returns the characters it was sent, as shown here.

```
>> char(fread(eeprom,16))'

ans =

Hello World!
```

7   Clean up by deleting and clearing the object.

```
fclose(eeprom);
delete(eeprom);
clear('eeprom');
```

NI-845x Example

This example shows how to communicate with an Analog Devices™ ADXL345 sensor chip on a circuit board, using an address of 53 hex and a board index of 0 on a NI-845x adaptor. In this case, the NI-845x adaptor board is plugged into the computer (via the USB port), and a circuit board containing the sensor chip is connected to the host adaptor board via wires. Note that the circuit has external pullups, as the NI-8451 adaptor used in this example does not have internal pullups.

To communicate with a sensor chip:

1   Ensure that the NI-845x adaptor is installed so that you can use the `i2c` interface.

```
>> instrhwinfo('i2c')

ans =

    InstalledAdaptors: {'Aardvark'  'NI845x'}
        JarFileVersion: 'Version 3.4'
```

2   Look at the NI-845x adaptor properties.

```
>> instrhwinfo('i2c', 'NI845x')

ans =

              AdaptorDllName: [1x120 char]
           AdaptorDllVersion: 'Version 3.4'
                 AdaptorName: 'NI845x'
               BoardIdsInUse: [1x0 double]
            InstalledBoardIDs: 0
         DetectedBoardSerials: {'0180D47A (BoardIndex: 0)'}
       ObjectConstructorName: 'i2c('NI845x', BoardIndex, RemoteAddress);'
              VendorDllName: 'Ni845x.dll'
        VendorDriverDescription: 'National Instruments NI USB 845x Driver'
```

Make sure that you have the NI-845x software driver installed. For details, see "I2C Interface Usage Requirements and Guidelines" on page 9-17.

3   Create the I2C object called `i2cobj`, using these properties:

```
% Vendor = NI845x
% BoardIndex = 0
% RemoteAddress = 53h

i2cobj = i2c('NI845x', 0, '53h');
```

You must provide these three parameters to create the object. Read the documentation of the chip in order to know what the remote address is.

Tip You can also see what the remote address of the chip is by scanning for instruments in the Test & Measurement Tool. In the tool, right-click the **I2C** node and select **Scan for I2C adaptors**. Any chips found by the scan is listed in the hardware tree. The listing includes the remote address of the chip.

---

4   Connect to the chip.

```
fopen(i2cobj)
```

5   Write to the sensor chip. Read the documentation or data sheet of the chip in order to know what the remote address is and other information about the chip. Usually chip manufacturers provide separate read and write addresses. The adaptor boards

only take one address (the read address) and handle conversions to read and write addresses.

In this case, the chip's device ID register is at address 0, so you need to write a 0 to the chip indicating you would like to read or write to the register.

```
fwrite(i2cobj, 0)
```

6 Read data back from the chip's device ID register using the fread function. Reading 1 byte of data returns the device ID registry. In the case of this chip, the read-only device ID register value is 229. Therefore, that is what is returned when you send the byte.

```
fread(i2cobj, 1)

ans =

    229
```

7 Clean up by deleting and clearing the object.

```
fclose(i2cobj);
delete(i2cobj);
clear('i2cobj');
```

Other Functionality

You can use these functions with the i2c object.

Function	Purpose
fopen	Connect interface object to instrument.
fread	Read binary data from instrument.
fwrite	Write binary data to instrument.
methods	Names and descriptions of functions that can be used with i2c objects.
fclose	Disconnect interface object from instrument.
record	Record data and event information to file.
propinfo	Display instrument object property information.

For more information about these functions, see the functions documentation.

Using Properties on an I2C Object

You can use the `get` function on the `i2c` object to see the available properties. In the first example shown in "Transmitting Data Over the I2C Interface" on page 9-9, which uses the Aardvark board, the syntax would be:

`get(eeprom)`

The following shows the output of the `get` from that example.

```
I2C specific properties:
BitRate = 100
BoardIndex = 0
BoardSerial = 2.23748e+09
PullupResistors = both
RemoteAddress = 80
TargetPower = both
Vendor = aardvark
```

For the example using a NI-845x board, as shown in the NI-845x section of "Configuring I2C Communication" on page 9-4, you see the following output.

`get(i2cobj)`

```
I2C specific properties:
BitRate = 100
BoardIndex = 0
BoardSerial = 0180D47A
PullupResistors = both
RemoteAddress = 0
Vendor = NI845x
```

Interface-specific properties that can be used with the `i2c` object include:

Property	Description
BitRate	Must be a positive, nonzero value specified in kHz. The adaptor and chips determine the rate. The default is 100 kHz for both the Aardvark and NI-845x adaptors.
TargetPower	Aardvark only. Can be specified as none or both. The value both means to power both lines, if supported. The value none means power no lines, and is the default value.
PullupResistors	Can be specified as none or both. The value both enables 2k pullup resistors to protect hardware in the I2C device, if supported. This is the default value. Devices may differ in their use of pullups. The Aardvark adaptor and the NI-8452 adaptor have internal pullup resistors to tie both bus lines to VDD and can be programmatically set. The NI-8451 does not have internal pullup resistors that can be programmatically set, and so require external pullups. Consult your device documentation to ensure that the correct pullups have been used.
BoardSerial	Unique identifier of the I2C master communication device.
Vendor	Use to create i2c object. Must be set to aardvark, for use with Aardvark adaptor, or NI845x for use with the NI-845x adaptor.
BoardIndex	Use to create i2c object. Specifies the board index of the hardware. Usually set to 0.
RemoteAddress	Use to create i2c object. Specifies the remote address of the hardware. Specified as a string when you create the i2c object. For example, to specify the remote address of 50 hex, use '50h'. Read the documentation of the chip in order to know what the remote address is. You can also see what the remote address of the chip is by scanning for instruments in the Test & Measurement Tool. In the tool, right-click the **I2C** node and select **Scan for I2C adaptors**. Any chips found by the scan is listed in the hardware tree. The listing includes the remote address of the chip.

All the I2C interface-specific properties work for both adaptor boards, except for `TargetPower`, which is Aardvark only.

The properties `Vendor`, `BoardIndex`, and `RemoteAddress` are used when you create the object, as shown in "Transmitting Data Over the I2C Interface" on page 9-9. The property `BoardSerial` is read-only. The `BitRate`, `TargetPower`, and `PullupResistors` properties can be set at any time after the object is created.

After you create the I2C object, you can set properties on it, as follows:

```
i2cobj.BitRate = 75
```

In this case, `i2cobj` is the name of the object, and you are changing the `BitRate` from the default of 100 kHz to 75 kHz.

The other two properties that you can set are strings, so they would be set as follows:

```
i2cobj.'TargetPower' = 'both'
```

In this case, `i2cobj` is the name of the object, and you are changing the `TargetPower` from the default of none to both. Note that `TargetPower` is only available using the Aardvark board, and does not apply to the NI-845x board.

I2C Interface Usage Requirements and Guidelines

The I2C interface does not support asynchronous behavior. Therefore, functions such as `fprintf`, `fscanf`, and `query` do not work. Use `fread` and `fwrite` to communicate using this interface.

You need to have either a Total Phase Aardvark host adaptor or a NI-845x adaptor board installed to use the `i2c` interface. The following sections describe requirements for each option.

Aardvark-specific Requirements

To use the I2C interface with the Aardvark adaptor, you must download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the Aardvark driver installed, see "Install the Total Phase Aardvark I2C/SPI Support Package" on page 14-9 to install it via the Support Package Installer.

You must install the Aardvark Software API and Share Library appropriate for your operating system.

The aardvark.dll file that comes with the Total Phase Aardvark adaptor board must be available in one of the following locations for use on Windows platforms.

- The location where MATLAB was started from (Bin folder).
- The MATLAB current folder (PWD).
- The Windows folder `C:\winnt` or `C:\windows`.
- The folders listed in the PATH environment variable.

The aardvark.so file that comes with the Total Phase Aardvark adaptor board must be in your MATLAB path for use on Linux platforms.

If you repower your Aardvark board, set the GPIO pins to output to get communication with a device to work. By default they are configured as input.

NI-845x-specific Requirements

To use the I2C interface with the NI-845x adaptor, you must download the Hardware Support Package to obtain the latest driver, if you do not already have driver

installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the NI-845x driver installed, see "Install the NI-845x I2C/SPI Interface Support Package" on page 14-8 to install it via the Support Package Installer.

Devices may differ in their use of pullups. The NI-8452 has internal pullup resistors to tie both bus lines to VDD and can be programmatically set. The NI-8451 does not have internal pullup resistors that can be programmatically set, and so require external pullups. Consult your device documentation to ensure that the correct pullups have been used.

10

Controlling Instruments Using SPI

SPI Interface Overview

In this section...
"SPI Communication" on page 10-2
"Supported Platforms for SPI" on page 10-2

SPI Communication

SPI, or Serial Peripheral Interface, is a synchronous serial data link standard that operates in full duplex mode. It is commonly used in the test and measurement field. Typical uses include communicating with micro controllers, EEPROMs, A2D devices, embedded controllers, etc.

Instrument Control Toolbox SPI support lets you open connections with individual chips and to read and write over the connections to individual chips using an Aardvark or NI-845x host adaptor.

The primary uses for the `spi` interface involve the `write`, `read`, and `writeAndRead` functions for synchronously reading and writing binary data. To identify SPI devices in Instrument Control Toolbox, use the `instrhwinfo` function on the SPI interface, called `spi`.

Supported Platforms for SPI

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface. The following sections contain the supported platforms for each option.

Using Aardvark

The SPI interface is supported on these platforms when used with the Aardvark host adaptor:

- Linux — Red Hat® Enterprise Linux 4 and 5 with kernel 2.6, and possibly SUSE® and Ubuntu distributions.

- Mac OS X 64-bit — Intel® versions of Mac OS X 10.5 Leopard, 10.6 Snow Leopard, 10.7 Lion, and 10.8 Mountain Lion.

- Microsoft Windows 32-bit and 64-bit — Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), Windows 7 (32-bit and 64-bit), and Windows 8 (32-bit and 64-bit).

Using NI-845x

The SPI interface is supported on these platforms when used with the NI-845x host adaptor:

- Microsoft Windows 32-bit and 64-bit — Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), Windows 7 (32-bit and 64-bit), and Windows 8 (32-bit and 64-bit).

Configuring SPI Communication

You must have a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface.

Configuring Aardvark

To use the SPI interface with the Aardvark adaptor, you must download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the Aardvark driver installed, see "Install the Total Phase Aardvark I2C/SPI Support Package" on page 14-9 to install it via the Support Package Installer.

The `aardvark.dll` file that comes with the Total Phase Aardvark adaptor board must be available in one of these locations for use on Windows platforms:

- The location where MATLAB was started from (bin folder)
- The MATLAB current folder (PWD)
- The Windows folder `C:\winnt` or `C:\windows`
- The folders listed in the path environment variable

Ensure that the Aardvark adaptor is installed properly.

```
>> instrhwinfo('spi')

ans =

    SupportedVendors: {'aardvark'}
    InstalledVendors: {'aardvark'}
```

Look at the adaptor properties.

```
>> instrhwinfo('spi' , 'Aardvark')

ans =

            VendorName: 'aardvark'
     VendorDescription: 'Total Phase I2C/SPI Driver'
     VendorLibraryName: 'aardvark.dll'
     InstalledBoardIds: {[0]}
    BoardSerialNumbers: {'2237722838'}
     ObjectConstructors: {'spi('aardvark', 0, 0)'}
```

Create a SPI object using the `spi` function. This example uses a SPI object called `S` that communicates to an EEPROM chip. Create the object using the `BoardIndex` and `Port` numbers, which are `0` in both cases.

```
S = spi('aardvark', 0, 0);
```

Display the object properties.

```
>> disp(S)
SPI Object :

    Adapter Settings
        BoardIndex:         0
        BoardSerial:        2237722838
        VendorName:         aardvark

    Communication Settings
        BitRate:            1000000 Hz
        ChipSelect:         0
        ClockPhase:         FirstEdge
        ClockPolarity:      IdleLow
        Port:               0

    Communication State
        ConnectionStatus:   Disconnected

    Read/Write State
        TransferStatus:     Idle
```

The communication settings properties reflect what was used to create the object – `BoardIndex` of `0` and `Port` of `0`. For information about other properties, see "Using Properties on the SPI Object" on page 10-18.

Configuring NI-845x

To use the SPI interface with the NI-845x adaptor, download the hardware support package to obtain the latest driver, if you do not already have the driver installed.

If you do not have the NI-845x driver installed, see "Install the NI-845x I2C/SPI Interface Support Package" on page 14-8 to install it via the Support Package Installer.

Ensure that the NI-845x adaptor is installed properly.

```
>> instrhwinfo('spi')

ans =

  HardwareInfo with properties:

    SupportedVendors: {'aardvark'  'ni845x'}
    InstalledVendors: {'ni845x'}
```

Look at the NI-845x adaptor properties.

```
>> instrhwinfo('spi', 'ni845x')

ans =

  HardwareInfo with properties:

              VendorName: 'ni845x'
       VendorDescription: 'National Instruments'
       VendorLibraryName: 'Ni845x.dll'
        InstalledBoardIds: {[0]}
       BoardSerialNumbers: {'0183388B'}
       ObjectConstructors: {'spi('ni845x', 0, 0)'}
```

Create a SPI object using the `spi` function. The example in the next section uses a SPI object called `s2` that communicates with an EEPROM chip. Create the object using the `BoardIndex` and `Port` numbers, which are `0` in both cases.

```
s2 = spi('ni845x', 0, 0);
```

Display the object properties.

```
>> disp(s2)
SPI Object :

    Adapter Settings
        BoardIndex:             0
        BoardSerial:            0183388B
        VendorName:             ni845x

    Communication Settings
        BitRate:                1000000 Hz
        ChipSelect:             0
        ClockPhase:             FirstEdge
        ClockPolarity:          IdleLow
        Port:                   0

    Communication State
        ConnectionStatus:       Disconnected

    Read/Write State
        TransferStatus:         Idle
```

The communication settings properties reflect what was used to create the object –
BoardIndex of 0 and Port of 0. For information about other properties, see "Using
Properties on the SPI Object" on page 10-18.

Transmitting Data Over the SPI Interface

The typical workflow for transmitting data over the SPI interface involves adaptor discovery, connection, communication, and cleanup. Discovery can be done only at the adaptor level. You must have a Total Phase Aardvark adaptor or an NI-845x adaptor board installed to use the `spi` interface.

Transmit Data Over SPI Using Aardvark

This example shows how to communicate with an EEPROM chip on a circuit board, with a board index of 0 and using port 0.

1   Ensure that the Aardvark adaptor is installed so that you can use the `spi` interface.

```
>> instrhwinfo('spi')

ans =

    SupportedVendors: {'aardvark'}
    InstalledVendors: {'aardvark'}
```

2   Look at the adaptor properties.

```
>> instrhwinfo('spi' , 'Aardvark')

ans =

            VendorName: 'aardvark'
     VendorDescription: 'Total Phase I2C/SPI Driver'
     VendorLibraryName: 'aardvark.dll'
      InstalledBoardIds: {[0]}
    BoardSerialNumbers: {'2237722838'}
    ObjectConstructors: {'spi('aardvark', 0, 0)'}
```

Make sure that you have the Aardvark software driver installed and that the `aardvark.dll` is on your MATLAB path. For details, see "SPI Interface Usage Requirements and Guidelines" on page 10-22.

3   Create the SPI object called `S`, using these properties:

```
% Vendor = aardvark
% BoardIndex = 0
% Port = 0

S = spi('aardvark', 0, 0);
```

You must provide these three parameters to create the object.

4   Look at the object properties.

```
>> S

S =

SPI Object :

    Adapter Settings
        BoardIndex:          0
        BoardSerial:         2237722838
        VendorName:          aardvark

    Communication Settings
        BitRate:             1000000 Hz
        ChipSelect:          0
        ClockPhase:          FirstEdge
        ClockPolarity:       IdleLow
        Port:                0

    Communication State
        ConnectionStatus:    Disconnected

    Read/Write State
        TransferStatus:      Idle
```

When you create the spi object, default communication settings are used, as shown here. To change any of these settings, see "Using Properties on the SPI Object" on page 10-18 for more information and a list of the properties.

5 Connect to the chip.

```
connect(S);
```

6 Read and write to the chip.

```
% Create a variable containing the data to write
dataToWrite = [3 0 0 0];
```

```
% Write the binary data to the chip
write(S, dataToWrite);

% Create a variable to contain 5 bytes of returned data
numData = 5

% Read the binary data from the chip
read(S, numData)
```

7 Disconnect the SPI device and clean up by clearing the object.

```
disconnect(S);
clear(S);
```

Transmit Data Over SPI Using NI-845x

This example shows how to communicate with an EEPROM chip on a circuit board, with a board index of 0 and using port 0.

1 Ensure that the NI-845x adaptor is installed so that you can use the `spi` interface.

```
>> instrhwinfo('spi')

ans =

  HardwareInfo with properties:

    SupportedVendors: {'aardvark'  'ni845x'}
    InstalledVendors: {'ni845x'}
```

2 Look at the NI-845x adaptor properties.

```
>> instrhwinfo('spi', 'ni845x')

ans =

  HardwareInfo with properties:

              VendorName: 'ni845x'
       VendorDescription: 'National Instruments'
       VendorLibraryName: 'Ni845x.dll'
       InstalledBoardIds: {[0]}
      BoardSerialNumbers: {'0183388B'}
      ObjectConstructors: {'spi('ni845x', 0, 0)'}
```

Make sure that you have the NI-845x software driver installed. For details, see "SPI Interface Usage Requirements and Guidelines" on page 10-22.

3 Create the SPI object called `s2`, using these properties:

```
% Vendor = ni845x
% BoardIndex = 0
% Port = 0

s2 = spi('ni845x', 0, 0);
```

You must provide these three parameters to create the object.

4 Look at the object properties.

```
>> disp(s2)
SPI Object :

    Adapter Settings
        BoardIndex:             0
        BoardSerial:            0183388B
        VendorName:             ni845x

    Communication Settings
        BitRate:                1000000 Hz
        ChipSelect:             0
        ClockPhase:             FirstEdge
        ClockPolarity:          IdleLow
        Port:                   0

    Communication State
        ConnectionStatus:       Disconnected

    Read/Write State
        TransferStatus:         Idle
```

When you create the `spi` object, default communication settings are used, as shown here. To change any of these settings, see "Using Properties on the SPI Object" on page 10-18 for more information and a list of the properties.

5 Connect to the chip.

```
connect(s2);
```

6 Read and write to the chip.

```
% Create a variable containing the data to write
dataToWrite = [3 0 0 0];

% Write the binary data to the chip
write(s2, dataToWrite);

% Create a variable to contain 5 bytes of returned data
```

```
numData = 5

% Read the binary data from the chip
read(s2, numData)

ans =

     0     0     0     0     0
```

7 Disconnect the SPI device and clean up by clearing the object.

```
disconnect(s2);
clear(s2);
```

SPI Functions

You can use these functions with the `spi` object.

Note: SPI is a full duplex communication protocol, and data must be written in order to read data. You can use the `read` function to write dummy data to the device. The `write` function flushes the data returned by the device. The `writeAndRead` function does the read and write together.

Function	Purpose
instrhwinfo	Check that the Aardvark and/or NI-845x adaptor is installed. `instrhwinfo('spi')` Look at the adaptor properties. `instrhwinfo('spi', 'Aardvark')` `instrhwinfo('spi', 'ni845x')`
spiinfo	Returns information about devices and displays the information on a per vendor basis.

Function	Purpose
	```info = spiinfo()```
connect	Connect the SPI object to the device. Use this syntax: ```connect(spiObject);```

read	Synchronously read binary data from the device. To read data, first create a variable, such as numData, to specify the size of the data to read. In this case, create the variable to read 5 bytes. Then use the read function as shown here, where spiObject is the name of your object. This process is also shown in step 6 of the previous example. The precision of the data is UINT8.
	
	```numData = 5;```
	
	```read(spiObject, numData);```
	
	Or you can use this syntax:
	
	```A = read(spiObject, size)```

write	Synchronously write binary data to the device. To write data, first create a variable, such as dataToWrite. In this case, create the data [3 0 0 0]. Then use the write function as shown here, where spiObject is the name of your object. This process is also shown in step 6 of the previous example. The precision of the data written is UINT8.
	
	```dataToWrite = [3 0 0 0];```
	
	```write(spiObject, dataToWrite);```

Function	Purpose
writeAndRead	Synchronously do a simultaneous read and write of binary data with the device. In this case, the function synchronously writes the data specified by the variable dataToWrite to the device in binary format, then synchronously reads from the device and returns the data to the variable data, as shown here, where spiObject is the name of your object. The precision of the data written and read is UINT8. dataToWrite = [3 0 0 0]; data = writeAndRead(spiObject, dataToWrite)
disconnect	Disconnect SPI object from the device. Use this syntax: disconnect(spiObject);

Using Properties on the SPI Object

Use the `properties` function on the `spi` object to see the available properties. In the preceding example, the syntax would be:

```
properties(S)
```

The following shows the output of the properties from the preceding example, "Transmitting Data Over the SPI Interface" on page 10-9.

```
>> properties(S)

Properties for class instrument.interface.spi.aardvark.Spi:

    BitRate
    ClockPhase
    ClockPolarity
    ChipSelect
    Port
    BoardIndex
    VendorName
    BoardSerial
    ConnectionStatus
    TransferStatus
```

You can use these interface-specific properties with the spi object.

Property	Description
BitRate	SPI clock speed. Must be a positive, nonzero value specified in Hz. The default is 1000000 Hz for both the Aardvark and NI-845x adaptors. To change from the default: `S.BitRate = 400000`
ClockPhase	SPI clock phase. Can be specified as 'FirstEdge' or 'SecondEdge'. The default of 'FirstEdge' is used if you do not specify a phase. ClockPhase indicates when the data is sampled. If set to 'FirstEdge', the first edge of the clock is used to sample the first data byte. The first edge may be the rising edge (if ClockPolarity is set to 'IdleLow'), or the falling edge (if ClockPolarity is set to 'IdleHigh'). If set to 'SecondEdge', the second edge of the clock is used to sample the first data byte. The second edge may be the falling edge (if ClockPolarity is set to 'IdleLow'), or the rising edge (if ClockPolarity is set to 'IdleHigh'). To change from the default: `S.ClockPhase = 'SecondEdge'`
ClockPolarity	SPI clock polarity. Can be specified as 'IdleLow' or 'IdleHigh'. The default of 'IdleLow' is used if you do not specify a phase. ClockPolarity indicates the level of the clock signal when idle. 'IdleLow' means the clock idle state is low, and 'IdleHigh' means the clock idle state is high. To change from the default: `S.Polarity = 'IdleHigh'`

Property	Description
ChipSelect	SPI chip select line. The Aardvark adaptor uses 0 as the chip select line since it has only one line, so that is the default and only valid value.
Port	Use to create spi object. Port number of your hardware, specified as the number 0. The Aardvark adaptor uses 0 as the port number when there is one adaptor board connected. If there are multiple boards connected, they could use ports 0 and 1. Specify port number as the third argument when you create the spi object: `S = spi('aardvark', 0, 0);`
BoardSerial	Unique identifier of the SPI communication device.
VendorName	Use to create spi object. Adaptor board vendor, must be set to `'aardvark'`, for use with Total Phase Aardvark adaptor or `'ni845x'` for use with the NI-845x adaptor. Specify the vendor as the first argument when you create the spi object: `S = spi('aardvark', 0, 0);`
BoardIndex	Use to create spi object. Specifies the board index of the hardware. Usually set to 0. Specify board index as the second argument when you create the spi object: `S = spi('aardvark', 0, 0);`
ConnectionStatus	Returns the connection status of the SPI object. Possible values are Disconnected (default) and Connected.

Property	Description
TransferStatus	Returns the read/write operation status of the SPI object. Possible values: Idle (default) – The device is not transferring any data. Read – The device is reading data. Write – The device is writing data. ReadWrite – The device is reading and writing data.

The properties all have defaults, as indicated in the table. You do not need to set a property unless you want to change it to a different value from the default. Aside from the three properties required to construct the object – VendorName, BoardIndex, and Port – any other property is set using the .dot notation syntax:

```
<object_name>.<property_name> = <value>
```

Here are a few examples of using this syntax.

```
Change the BitRate from the default of 1000000 to 500000 kHz

S.BitRate = 500000

Change the ClockPhase from the default of 'FirstEdge' to 'SecondEdge'

S.ClockPhase = 'SecondEdge'
```

where S is the name of the object used in the examples.

SPI Interface Usage Requirements and Guidelines

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface. The following sections describe requirements for each option.

Aardvark-specific Requirements

To use the SPI interface with the Aardvark adaptor, you must download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the Aardvark driver installed, see "Install the Total Phase Aardvark I2C/SPI Support Package" on page 14-9 to install it via the Support Package Installer.

Install the Aardvark Software API and Shared Library appropriate for your operating system.

The `aardvark.dll` file that comes with the Total Phase Aardvark adaptor board must be available in one of these locations for use on Windows platforms:

- Location where MATLAB was started from (bin folder)
- MATLAB current folder (PWD)
- Windows folder `C:\winnt` or `C:\windows`
- Folders listed in the path environment variable

For use on Linux platforms, the `aardvark.so` file that comes with the Total Phase Aardvark adaptor board must be in your MATLAB path.

NI-845x-specific Requirements

To use the SPI interface with the NI-845x adaptor, you must download the hardware support package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this support package.

If you do not have the NI-845x driver installed, see "Install the NI-845x I2C/SPI Interface Support Package" on page 14-8 to install it via the Support Package Installer.

Using Device Objects

This chapter describes specific features and actions related to using device objects.

Device Objects

In this section...

Overview

All instruments attached to your computer must communicate through an interface. Popular interface protocols include GPIB, VISA, RS-232 (serial), and RS-485 (serial). While Instrument Control Toolbox interface objects allow you to communicate with your equipment at a low (instrument command) level, Instrument Control Toolbox also allows you to communicate with your equipment without detailed knowledge of how the hardware interface operates.

Programmable devices understand a specific language, sometimes referred to as its command set. One common set is called SCPI (Standard Commands for Programmable Instruments).

Device objects allow you to configure and query an instrument without knowledge of its command set. This 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 11-5. If you want a high-level description of all the steps you are likely to take when communicating with your instrument, refer to the Getting Started documentation that is linked to from the top of the main Instrument Control Toolbox Doc Center page.

What Are Device Objects?

Device objects are used to represent instruments in MATLAB workspace. Properties and methods specific to an instrument are encapsulated within device objects. Device objects also free you from the specific underlying commands required to communicate with your hardware.

You can use device objects at the MATLAB Command Window, inside functions, scripts, and graphical user interface callbacks. The low-level communication is performed through a MATLAB instrument driver.
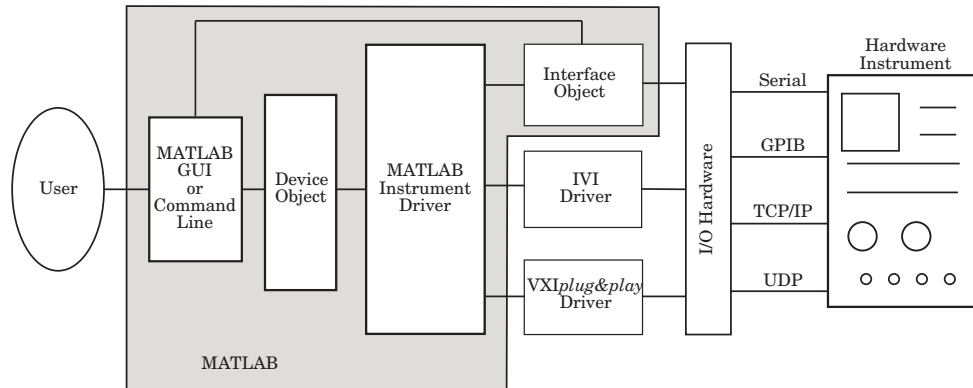
Device Objects for MATLAB Instrument Drivers

There are three types of MATLAB instrument drivers:

- MATLAB interface instrument driver
- MATLAB IVI instrument driver
- MATLAB VXI*plug&play* instrument driver
- Generic instrument driver

Instrument Control Toolbox device objects support all these types of MATLAB drivers, so that by using a device object, you can interface with any of these drivers in the same way. However, each of these drivers interfaces differently with the hardware. While MATLAB IVI and MATLAB VXI*plug&play* drivers interface directly through standard drivers and the hardware port to the instrument, the MATLAB interface driver requires an interface object to communicate with the instrument. You can use generic drivers to communicate with devices or software. For more information on generic drivers, see "Generic Drivers: Overview" on page 15-2.

The Instrument Control Toolbox software supports the following interface objects:

- gpib
- serial

- `tcpip`
- `udp`
- `visa`

To learn how to create and use interface objects, see "Creating an Interface Object" on page 3-2.

Note If you are using an interface object with a device object and a MATLAB interface driver, you do not need to connect the interface object to the interface using the `fopen` command. You need to connect the device object only.

Available MATLAB Instrument Drivers

Several drivers ship with the Instrument Control Toolbox software. You can find these drivers by looking in the directory

matlabroot`\toolbox\instrument\instrument\drivers`

where *matlabroot* is the MATLAB installation directory, as seen when you type

`matlabroot`

at the MATLAB Command Window.

Many other drivers are available on the MathWorks Web site at

`http://www.mathworks.com/matlabcentral/fileexchange`

including drivers specifically for the Instrument Control Toolbox software.

Creating and Connecting Device Objects

In this section...

Device Objects for MATLAB Interface Drivers

Create a MATLAB device object to communicate with a Tektronix TDS 210 Oscilloscope. To communicate with the scope you will use a National Instruments GPIB controller.

1 First create an interface object for the GPIB hardware. The following command creates a GPIB object for a National Instruments GPIB board at index 0 with an instrument at primary address 1.

```
g = gpib('ni',0,1);
```
2 Now that you have created the interface object, you can construct a device object that uses it. The command to use is `icdevice`. You need to supply the name of the instrument driver, `tektronix_tds210`, and the interface object created for the GPIB controller, `g`.

```
d = icdevice('tektronix_tds210', g);
```

You can use the `whos` command to display the size and class of `d`.

```
whos d
  Name      Size                    Bytes  Class

  d         1x1                       652  icdevice object

Grand total is 22 elements using 652 bytes
```

Device Object Properties

A device object has a set of base properties and a set of properties defined by the driver. All device objects have the same base properties, regardless of the driver being used. The driver properties are defined by the driver specified in the `icdevice` constructor.

Device Object Display

Device objects provide you with a convenient display that summarizes important object information. You can invoke the display in these ways:

- Type the name of the device object at the command line.
- Exclude the semicolon when creating the device object.
- Exclude the semicolon when configuring properties using dot notation.
- Pass the object to the disp or display function.

The display summary for device object d is given below.

```
Instrument Device Object Using Driver : tektronix_tds210.mdd

   Instrument Information
     Type:              Oscilloscope
     Manufacturer:      Tektronix
     Model:             TDS210

   Driver Information
     DriverType:        MATLAB interface object
     DriverName:        tektronix_tds210.mdd
     DriverVersion:     1.0

   Communication State
     Status:            open
```

You can also display summary information via the Workspace browser by right-clicking a device object and selecting **Display Summary** from the context menu.

Device Objects for VXI*plug&play* and IVI Drivers

Creating the MATLAB Instrument Driver

The command-line function makemid creates a MATLAB instrument driver from a VXI*plug&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 VXI*plug&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 VXI*plug&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

```
connect(d, 'instrument');
```

If `connect` is successful, the device object's status property is set to `open`; otherwise it remains as `closed`. You can check the status of this property with the `get` function or by looking at the object display.

```
d.status

ans =
```

```
open
```

Communicating with Instruments

In this section...

Configuring Instrument Settings

Once a device object has been created and connected, it can be used as the interface to an instrument. This chapter shows you how to access and configure your instrument's settings, as well as how to read and write data to the instrument.

Every device object contains properties specific to the instrument it represents. These properties are defined by the instrument driver used during device object creation. For example, there may be properties for an oscilloscope that allow you to adjust trigger parameters, or the contrast of the screen display.

Properties are assigned default values at device object creation. On execution of `connect` the object is updated to reflect the state of the instrument or vice versa, depending on the second argument given to `connect`.

You can obtain a full listing of configurable properties by calling the `set` command and passing the device object.

Configuring Settings on an Oscilloscope

This example illustrates how to configure an instrument using a device object.

The instrument used is a Tektronix TDS 210 two-channel oscilloscope. A square wave is input into channel 1 of the oscilloscope. The task is to adjust the scope's settings so that triggering occurs on the falling edge of the signal:

1   **Create the device object** — Create a GPIB interface object, and then a device object for a TDS 210 oscilloscope.

```
g = gpib('ni',0,1);
d = icdevice('tektronix_tds210', g);
```

2   **Connect the device object** — Use the `connect` function to connect the device object to the instrument.

```
connect(d);
```

3 **Check the current Slope settings for the Trigger property**— Create a variable to represent the Trigger property and then use the get function to obtain the current value for the oscilloscope Slope setting.

```
dtrigger = get(d, 'Trigger');
dtrigger.Slope
ans =

    rising
```

The Slope is currently set to rising.

4 **Change the Slope setting** — If you want triggering to occur on the falling edge, you need to modify that setting in the device object. This can be accomplished with the set command.

```
dtrigger.Slope = 'falling');
```

This changes Slope to falling.

5 **Disconnect and clean up** — When you no longer need the device object, disconnect it from the instrument and remove it from memory. Remove the device object and interface object from the MATLAB workspace.

```
disconnect(d);
delete(d);
clear d g dtrigger;
```

Calling Device Object Methods

Device objects contain methods specific to the instruments they represent. Implementation details are hidden behind a single function. Instrument-specific functions are defined in the MATLAB instrument driver.

The methods function displays all available driver-defined functions for the device object. The display is divided into two sections:

- Generic object functions
- Driver-specific object functions

To view the available methods, type

```
methods(obj)
```

Use the `instrhelp` function to get help on the device object functions.

```
instrhelp(obj, methodname);
```

To call instrument-specific methods you use the `invoke` function. `invoke` requires the device object and the name of the function. You must also provide input arguments, when appropriate. The following example demonstrates how to use `invoke` to obtain measurement data from an oscilloscope.

Using Device Object Functions

This example illustrates how to call an instrument-specific device object function. Your task is to obtain the frequency measurement of a waveform. The instrument is a Tektronix TDS 210 two-channel oscilloscope.

The scope has been preconfigured with a square wave input into channel 1 of the oscilloscope. The hardware supports four different measurements: frequency, mean, period, and peak-to-peak. The requested measurement is signified with the use of an index variable from 1 to 4.

For demonstration purposes, the oscilloscope in this example has been preconfigured with the correct measurement settings:

1 **Create the device object** — Create a GPIB interface object and a device object for the oscilloscope.

    ```
    g = gpib('ni',0,1);
    d = icdevice('tektronix_tds210', g);
    ```
2 **Connect the device object** — Use the `connect` command to open the GPIB object and update the settings in the device object.

    ```
    connect(d);
    ```
3 **Obtain the frequency measurement** — Use the `invoke` command and call `measure`. The `measure` function requires that an index parameter be specified. The value of the index specifies which measurement the oscilloscope should return. For the current setup of the Tektronix TDS 210 oscilloscope, an index of 1 indicates that frequency is to be measured.

    ```
    invoke(d, 'measure', 1)

    ans =
    ```

11-11

```
999.9609
```

The frequency returned is 999.96 Hz, or nearly 1 kHz.

4 **Disconnect and clean up** — You no longer need the device object so you can disconnect it from the instrument. You should also delete it from memory and remove it from the MATLAB workspace.

```
disconnect(d);
delete(d);
clear d g;
```

Control Commands

Control commands are special functions and properties that exist for all device objects. You use control commands to identify an instrument, reset hardware settings, perform diagnostic routines, and retrieve instrument errors. The set of control commands consists of

- "InstrumentModel" on page 11-12
- "devicereset" on page 11-12
- "selftest" on page 11-13
- "geterror" on page 11-13

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,

```
d.InstrumentModel
```

```
ans =
```

```
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v2.03 TDS2MM:MMV:v1.04
```

devicereset

To restore the factory settings on your instrument, use the `devicereset` function. When `devicereset` is called, the appropriate reset instruction is sent to your instrument.

The command accepts a connected device object and has no output arguments.

```
devicereset(obj);
```

selftest

This command requests that your instrument perform a self-diagnostic. The actual operations performed and output arguments are specific to the instrument your device object is connected to. `selftest` accepts a connected device object as an input argument.

```
result = selftest(obj);
```

geterror

You can retrieve error messages generated by your instrument with the `geterror` function. The returned messages are instrument specific. `geterror` accepts a connected device object as an input argument.

```
msg = geterror(obj);
```

Device Groups

Working with Group Objects

Device groups are used to group several related properties. For example, a channel group might contain the input channels of an oscilloscope, and the properties and methods specific to the input channels on the instrument.

MATLAB instrument drivers specify the type and quantity of device groups for device objects.

Group objects can be accessed via the `get` command. For the Tektronix TDS 210 oscilloscope, there is a channel group that contains two group objects. The device property to access a group is always the group name.

```
chans = get(d, 'Channel')
```

HwIndex:	HwName:	Type:	Name:
1	CH1	scope-channel	Channel1
2	CH2	scope-channel	Channel2

To display the functions that a device group object supports, use the `methods` function.

```
methods(chans(1))
```

You can also display a list of the group object's properties and their current settings.

```
chans(2)
```

To get help on a driver-specific property or function, use the `instrhelp` function, with the name of the function or property.

```
instrhelp(chans(1),'Coupling')
```

Using Device Groups to Access Instrument Data

This example shows how to obtain waveform data from a Tektronix TDS 210 oscilloscope with a square wave signal input on channel 1, on a Windows machine. The methods used are specific to this instrument:

1 **Create and connect** — First, create the device object for the oscilloscope and then connect to the instrument.

```
s = serial('com1');
d = icdevice('tektronix_tds210', s);
connect(d);
```

2 **Get the device group** — To retrieve waveform data, first gain access to the Waveform group for the device object.

```
w = d.waveform;
```

This group is specific for the hardware you are using. The TDS 210 oscilloscope has one Waveform; therefore the group contains one group object.

```
HwIndex:     HwName:      Type:              Name:
1            Waveform1    scope-waveform     Waveform1
```

3 **Obtain the waveform** — Now that you have access to the Waveform group objects, you can call the `readwaveform` function to acquire the data. For this example, channel 1 of the oscilloscope is reading the signal. To access this channel, call `readwaveform` on the first channel.

```
wave = invoke(w, 'readwaveform', 'channel1');
```

4 **View the data** — The `wave` variable now contains the waveform data from the oscilloscope. Use the `plot` command to view the data.

```
plot(wave);
```

5 **Disconnect and clean up** — Once the task is done, disconnect the hardware and free the memory used by the objects.

```
disconnect(d)
delete([d s])
clear d, s, w, wave;
```

Using VXI*plug&play* Drivers

This chapter describes the use of VXI*plug&play* drivers for instrument control. The sections are as follows.

- "VXI plug and play Setup" on page 12-2
- "VXI plug and play Drivers" on page 12-4

VXI plug and play Setup

Instrument Control Toolbox Software and VXI*plug&play* Drivers

The Instrument Control Toolbox software can communicate with hardware using VXI*plug&play* drivers. Most often, the instrument manufacturers supply these drivers.

For definitions and specifications of VXI*plug&play* drivers, see the Web site of the IVI Foundation at http://ivifoundation.org/specifications/default.aspx.

VISA Setup

A system must have VISA installed in order for VXI*plug&play* drivers to work. The driver installer software might specify certain VISA or other connectivity requirements.

To determine whether your system is properly configured with the necessary version of VISA, at the MATLAB Command window, type:

```
instrhwinfo visa
ans =
    InstalledAdaptors: {'agilent'}
       JarFileVersion: 'Version 2.0 (R14)'
```

The cell array returned for `InstalledAdaptors` indicates which VISA software is installed. A 1x0 cell array indicates that no VISA is installed. Possible `InstalledAdaptors` values are `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 VXI*plug&play* driver, or you might be able to download VISA from the instrument manufacturer's Web site.

Other Software Requirements

An instrument driver can have other software requirements in addition to or instead of VISA. Consult the driver documentation. The installer software itself might specify these requirements.

VXI plug and play Drivers

Installing VXI *plug&play* Drivers

The VXI*plug&play* driver particular to a piece of equipment is usually provided by the equipment manufacturer as either an installation disk or as a Web download. Once the driver is installed, you can determine whether the configuration is visible to MATLAB software by using the Test & Measurement Tool to view the current driver installations. Open the tool by typing:

```
tmtool
```

Expand the `Instrument Drivers` node and click `VXIplug&play Drivers`. Click the **Scan** button to update the display. All installed VXI*plug&play* drivers will be listed.
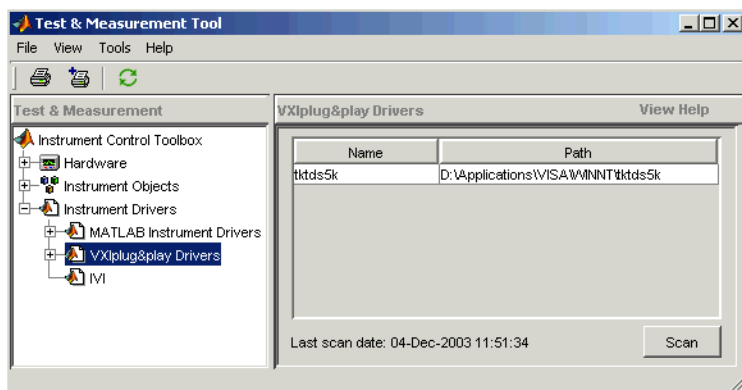
Alternatively, you can use the MATLAB function `instrhwinfo` to find out which drivers are installed.

```
instrhwinfo ('vxipnp')
```

```
ans =
    InstalledDrivers: {'tktds5k', 'ag3325b', 'hpe363xa'}
      VXIPnPRootPath: 'C:\VXIPNP\WINNT'
```

The cell array returned for `InstalledDrivers` contains the names of all the installed VXI*plug&play* drivers. The string returned for `VXIPnPRootPath` indicates where the drivers are installed.

Creating a MATLAB 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 the Instrument Control Toolbox software, at

http://www.mathworks.com/products/supportedio.html?prodCode=IC

On this page, click the **VXI***plug&play* 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 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 VXI*plug&play* instrument driver name (identified by `instrhwinfo`), and *filename* is the file containing the resulting MATLAB instrument driver. See the `makemid` reference page for details on this function.

If you need to customize the driver, open the new driver in the MATLAB Instrument Driver Editor, modify it as required, and save it.

Note When you create a MATLAB instrument driver based on a VXI*plug&play* driver, the original driver must remain installed on your system for you to use the new MATLAB instrument driver.

Importing with the MATLAB Instrument Driver Editor (midedit)

The MATLAB Instrument Driver Editor can import a VXI*plug&play* driver, thereby creating a MATLAB VXI*plug&play* instrument driver. You can evaluate or set the driver's functions and properties, and you can save the modified MATLAB instrument driver for further use:

1 Open the MATLAB Instrument Driver Editor with `midedit`.
2 Select **File > Import**.
3 In the Import Driver dialog box, select the VXI*plug&play* driver that you want to import and click **Import**.

 The MATLAB Instrument Driver Editor loads the driver and displays the components of the driver, as shown in the following figures.
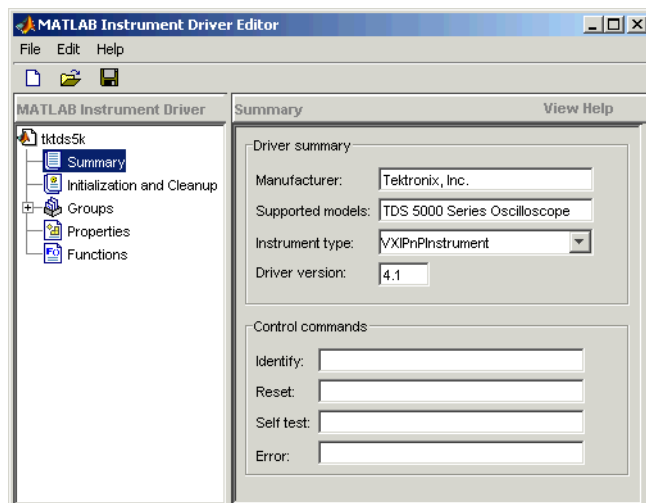
MATLAB Instrument Driver Editor Showing tktds5k MATLAB Instrument Driver Summary
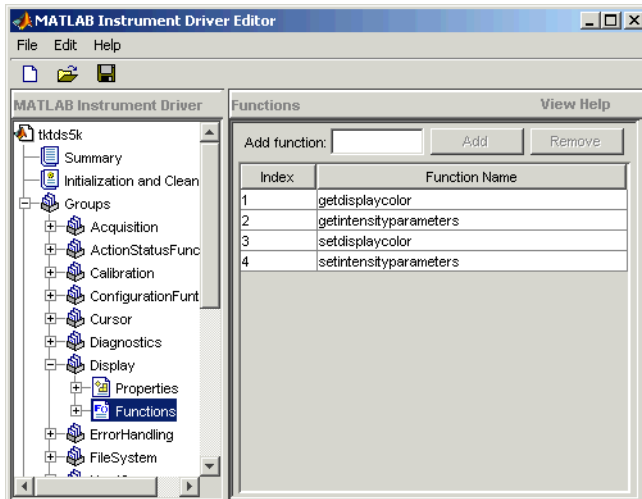
tktds5k MATLAB Instrument Driver Display Group Functions

With the MATLAB Instrument Driver Editor, you can:

- Create, delete, modify, and rename properties, functions, or groups.
- Add code around instrument commands for analysis.
- Add create, connect, and disconnect code.
- Save the driver as a MATLAB VXI*plug&play* instrument driver.

For more information, see "MATLAB Instrument Driver Editor Overview" on page 18-2.

Note When you create a MATLAB instrument driver based on a VXI*plug&play* driver, the original driver must remain installed on your system for you to use the new MATLAB instrument driver.

Constructing Device Objects Using a MATLAB 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:

```
obj = icdevice('MATLABVXIpnpDriver.mdd','GPIB0::2::INSTR')
connect(obj)
```

See the `icdevice` reference page for full details about this function.

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

When using IVI-C or VXI Plug&Play drivers, executing your code will generate additional file(s) in the folder specified by executing the following code at the MATLAB prompt:

```
sprintf('%s',[tempdir 'ICTDeploymentFiles'])
```

On all supported platforms, a file with the name
MATLABPrototypeFor<driverName>.m is generated, where <driverName> the name of the IVI-C or VXI Plug&Play driver. With 64-bit MATLAB on Windows, a second file by the name <driverName>_thunk_pcwin64.dll is generated. When creating your deployed application or shared library, manually include these generated files. For more information on including additional files refer to the MATLAB Compiler documentation.

Using IVI Drivers

This chapter describes the use of IVI drivers for instrument control.

IVI Drivers Overview

In this section...
"Instrument Control Toolbox Software and IVI Drivers" on page 13-2
"IVI-C and IVI-COM" on page 13-2

Instrument Control Toolbox Software and IVI Drivers

Instrument Control Toolbox software communicates with instruments using Interchangeable Virtual Instrument (IVI) drivers. The toolbox supports IVI-C and IVI-COM drivers, provided by various instrument manufacturers.

For definitions and specifications of IVI drivers and their components, see the IVI Foundation Web site at http://www.ivifoundation.org.

IVI-C and IVI-COM

Instrument Control Toolbox software supports IVI-C and IVI-COM drivers, with class-compliant and instrument-specific functionality.

IVI class-compliant drivers support common functionality across a family of related instruments. Use class-compliant drivers to access the basic functionality of an instrument, and the ability to swap instruments without changing the code in your application. With an IVI instrument-specific driver or interface, you can access the unique functionality of the instrument. The instrument-specific driver generally does not accommodate instrument substitution.

For IVI-C drivers, you can use IVI-C class drivers and IVI-C specific drivers. Device objects you construct to call IVI-C class drivers offer interchangeability between similar instruments, and work with all instruments consistent with that class driver. Device objects you construct to call IVI-C specific drivers directly generally offer less interchangeability, but provide access to the unique methods and properties of a specific instrument.

Instrument Interchangeability

Minimal Code Changes

With IVI class-compliant drivers, you can exchange instruments with minimal code changes. You can write your code for an instrument from one manufacturer and then swap it for the same type of instrument from another manufacturer. After editing the configuration file that identifies a new instrument, driver, and the hardware address, you can run your code without modifying it.

Effective Use of Interchangeability

To use the interchangeability of IVI effectively:

- Install drivers for both instruments of the same type (IVI-C or IVI-COM).
- Ensure that both drivers implement the same instrument class. For example, both must conform to the requirements for IviDmm or IviScope.
- When using IVI-C your program needs a Class Driver that instantiates the Class Compliant Specific Driver and calls class-compliant functions in it.
- Ensure that your program does not call instrument-specific functions.

You can enhance your code to handle the differences between the instruments or drivers you are using. You can still use these instruments interchangeably.

Examples of Interchangeability

The following diagram show interchangeability between instruments using IVI-COM and IVI-C drivers.
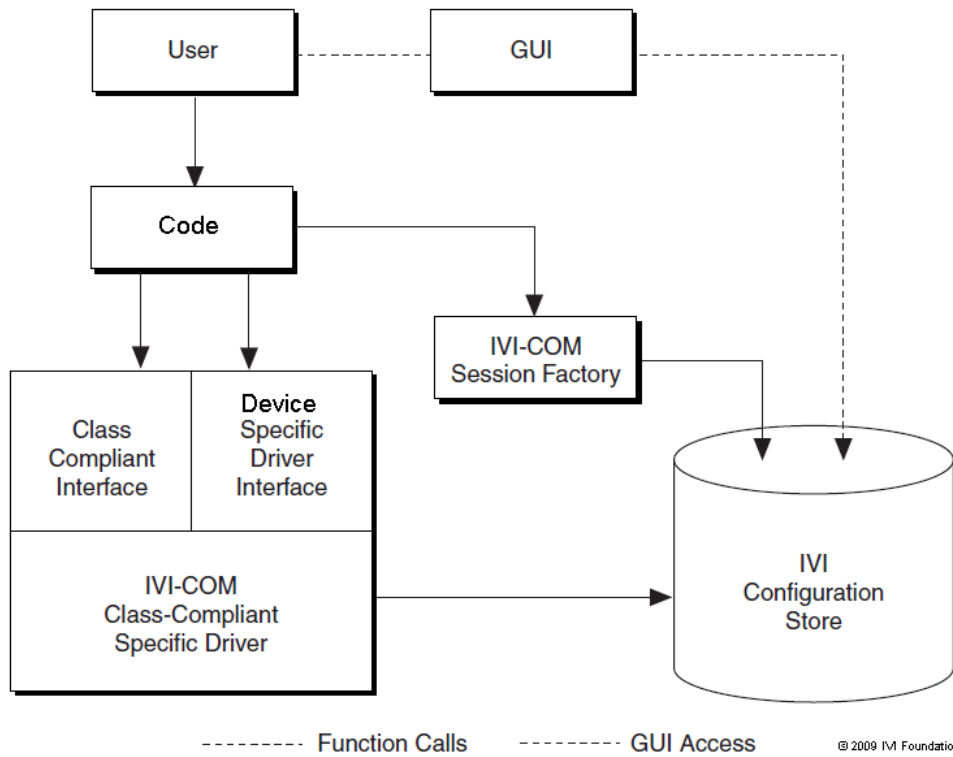
Using an IVI-COM Class Compliant Driver

Function Calls ---------- GUI Access © 2009 IVI Foundation
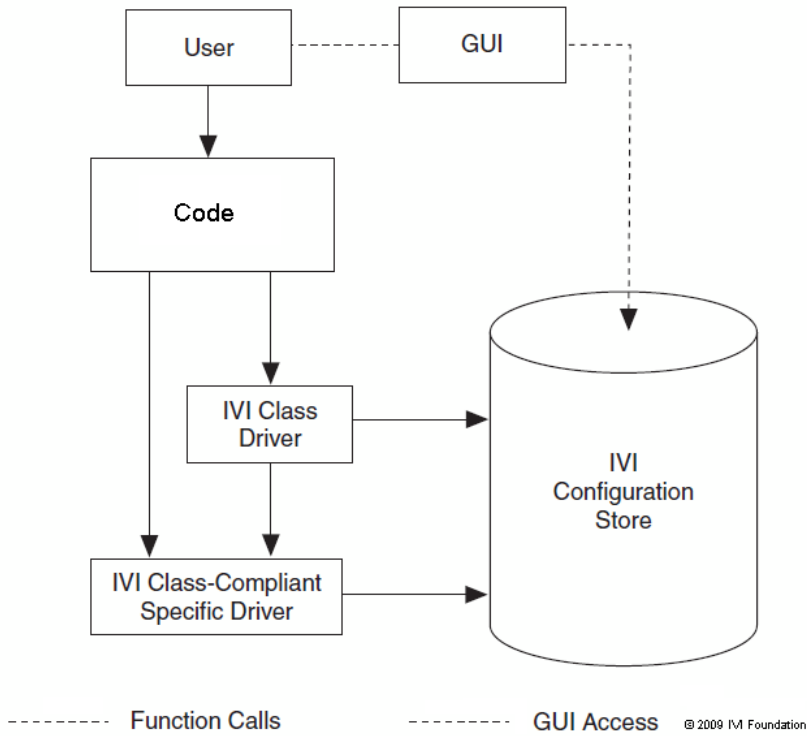
Using an IVI-C Class Compliant Driver

Getting Started with IVI Drivers

In this section...
"Introduction" on page 13-6
"Requirements to Work with MATLAB" on page 13-7
"Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI" on page 13-9
"MATLAB IVI Instrument Driver" on page 13-10
"Using MATLAB IVI Wrappers" on page 13-13

Introduction

You need to install IVI drivers and shared components before you can use them in MATLAB. See Requirements below for more information. You can use an IVI driver in MATLAB in two different ways. The syntax for each method differs vastly. Please refer to the MathWorks IVI Web page for more information. After installing the necessary components, you can:

- Create and use a MATLAB IVI instrument driver as described in MATLAB® IVI Instrument Driver. Here, you create a MATLAB IVI instrument driver with `.mdd` extension using an IVI driver.

- Use a MATLAB IVI wrapper as described in Using MATLAB® IVI Wrappers. Here, MATLAB wraps the IVI driver. You can then use this wrapper with the Instrument Control Toolbox software. This allows interchangeability and is the preferred method if you are working with class-compliant drivers.

  You can use the MATLAB IVI Wrappers provided with the Instrument Control Toolbox software with IVI drivers of the same class. Supported IVI driver classes are:

  - `IviACPwr`
  - `IviCounter`
  - `IviDCPwr`
  - `IviDigitizer`
  - `IviDmm`
  - `IviDownconverter`
  - `IviFgen`

- `IviPwrMeter`
- `IviUpconverter`
- `IviRFSigGen`
- `IviScope`
- `IviSpecAn`
- `IviSwtch`

You can also use MATLAB IVI wrappers provided by an instrument vendor that has built in MATLAB support. Refer to the vendor documentation for more information about using these drivers in MATLAB.

With the MATLAB IVI instrument driver, you construct a device object, which you use to communicate with your instrument. With the MATLAB IVI wrapper, you communicate with the instrument by directly accessing elements of the driver class.

Requirements to Work with MATLAB

Before you use IVI drivers in MATLAB, install:

- VISA
- IVI Shared components
- Required IVI drivers

Verifying VISA

Most IVI drivers require you to install VISA libraries on your system. The driver installer software specifies certain VISA or other connectivity requirements.

To determine proper configuration of the necessary version of VISA on your system, at the MATLAB Command Window, type:

```
instrhwinfo visa
ans =
    InstalledAdaptors: {'agilent'}
       JarFileVersion: 'Version 2.8.0''
```

The cell array returned for `InstalledAdaptors` indicates the type of VISA software installed. A 1-by-0 cell array indicates that your system does not have VISA installed. Possible `InstalledAdaptors` values are `agilent`, `tek`, and `ni`.

To install VISA, check the software installation disk provided with your instrument. This disk can include VISA along with the IVI driver for the instrument. You can also download VISA from the Web site of the instrument manufacturer.

An instrument driver can have other software requirements in addition to or instead of VISA. Consult the driver documentation. The installer software itself can specify these requirements.

Verifying IVI Shared Components

Many driver elements are common to a wide variety of instruments and not contained in the driver itself. You install them separately as *shared components*. Sharing components keeps the drivers as small and interchangeable as possible. You can use `instrhwinfo` to determine whether you installed shared components on your system.

```
instrhwinfo ('ivi')
ans =
.
.
.
ConfigurationServerVersion: '1.6.0.10124'
  MasterConfigurationStore: 'C:\Program Files\IVI\Data\IviConfigurationStore.xml'
               IVIRootPath: 'C:\Program Files\IVI\'
```

`ConfigurationServerVersion`, `MasterConfigurationStore`, and `IVIRootPath` all convey information related to installed shared components. `ConfigurationServerVersion` indicates whether you installed IVI shared components. If its value is an empty string, then you have not installed shared components.

Verifying IVI Drivers

The instrument manufacturer usually provides the specific IVI driver, either on an installation disk or as a Web download. Required VISA software and IVI shared components could also come with the driver.

You can use `instrhwinfo` to find information on installed IVI drivers and shared components.

```
instrhwinfo ('ivi')
ans =
                LogicalNames: {'MainScope', 'FuncGen'}
                  ProgramIDs: {'TekScope.TekScope','Agilent33250'}
                     Modules: {'ag3325b', 'hpe363xa'}
ConfigurationServerVersion: '1.6.0.10124'
  MasterConfigurationStore: 'C:\Program Files\IVI\Data\
                             IviConfigurationStore.xml'
```

```
IVIRootPath: 'C:\Program Files\IVI\'
```

Logical names are associated with particular IVI drivers, but they do not necessarily imply that the drivers are currently installed. You can install drivers that do not have a `LogicalName` property set yet, or drivers whose `LogicalName` was removed.

Alternatively, use the Test & Measurement Tool to view the installation of IVI drivers and the setup of the IVI configuration store. Open the tool by typing:

`tmtool`

Expand the `Instrument Drivers` node and click **IVI**. Click the **Software Modules** tab. (For information on the other IVI driver tabs and settings in the Test & Measurement Tool, see "IVI Configuration Store" on page 13-16.)

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

When using IVI-C or VXI Plug&Play drivers, executing your code will generate additional file(s) in the folder specified by executing the following code at the MATLAB prompt:

```
fullfile(tempdir,'ICTDeploymentFiles',sprintf('R%s',version('-release')))
```

On all supported platforms, a file with the name
MATLABPrototypeFor<driverName>.m is generated, where <driverName> the name
of the IVI-C or VXI Plug&Play driver. With 64-bit MATLAB on Windows, a second file
by the name <driverName>_thunk_pcwin64.dll is generated. When creating your
deployed application or shared library, manually include these generated files. For more
information on including additional files refer to the MATLAB Compiler documentation.

MATLAB IVI Instrument Driver

- "Using a MATLAB IVI Instrument Driver" on page 13-10
- "Creating a MATLAB IVI Instrument Driver with makemid" on page 13-10
- "Downloading a MATLAB IVI Instrument Driver" on page 13-11
- "Importing MATLAB IVI Instrument Drivers" on page 13-11
- "Constructing Device Objects Using a MATLAB IVI Instrument Driver" on page 13-12

Using a MATLAB IVI Instrument Driver

To use an IVI driver with a device object, you need a MATLAB IVI instrument driver
based upon the information in the original IVI driver. The MATLAB IVI instrument
driver, whether modified or not, acts as a wrapper to the IVI driver. These drivers,
however, do not support interchangeability. You can download or create the MATLAB
IVI instrument driver.

Creating a MATLAB IVI Instrument Driver with makemid

The command-line function makemid creates a MATLAB IVI instrument driver from an
IVI driver, saving the new driver in a file on disk. The syntax is:

```
makemid('driver','filename')
```

driver is the original IVI driver name (identified by instrhwinfo or the Test &
Measurement Tool), and *filename* is the MATLAB IVI instrument driver name. For
driver use a Module name(for IVI-C), a ProgramID (for IVI-COM), or a LogicalNames
value (for either IVI-C or IVI-COM). See the makemid reference page for full details on
this function.

To customize the driver, open the new driver in the MATLAB Instrument Driver Editor,
modify it as required, and save it.

Tip Do not uninstall the original IVI driver when you create a MATLAB IVI instrument driver based on an IVI driver. You need the IVI driver in order to use the new MATLAB IVI instrument driver.

Note: When you create a MATLAB IVI instrument driver without specifying an interface name, `makemid` uses the instrument-specific interface as the default interface.

Downloading a MATLAB IVI Instrument Driver

Go to the MATLAB Central Web site and search for an appropriate MATLAB IVI instrument driver for your instrument. You can look for wrappers using the **instrument drivers** tag in the File Exchange area.

To use the downloaded MATLAB IVI instrument driver, you also need the IVI driver for the installed instrument. Find this driver on the Web site of the instrument manufacturer.

Importing MATLAB IVI Instrument Drivers

You can import an IVI driver using the MATLAB Instrument Driver Editor, and create a MATLAB IVI instrument driver. Evaluate or set the functions and properties of the driver, and save the modified MATLAB IVI instrument driver for further use.

1   Open the MATLAB Instrument Driver Editor by typing `midedit`.
2   Select **File > Import**. The Import Driver dialog box opens.
3   Select the IVI driver that you want to import, and click **Import**.

    The MATLAB Instrument Driver Editor loads the driver and displays its components.
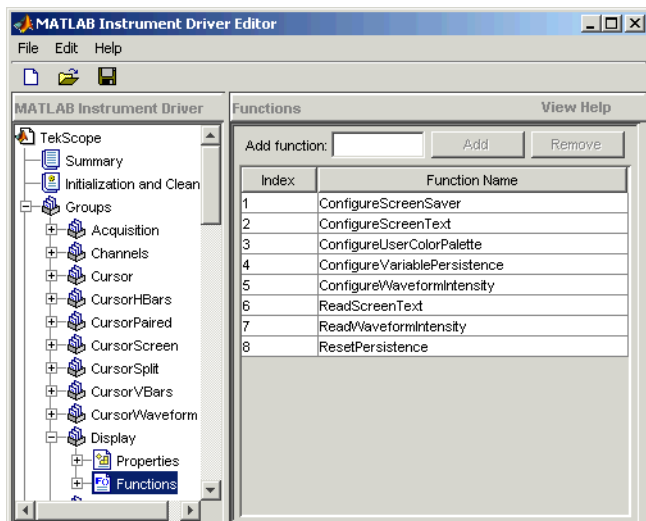
With the MATLAB Instrument Driver Editor, you can do the following:

- Create, delete, modify, and rename properties, functions, or groups.
- Add code around instrument commands for analysis.
- Add, create, connect, and disconnect code.
- Save the driver as a MATLAB IVI instrument driver.

For more information, see "MATLAB Instrument Driver Editor Overview" on page 18-2.

Tip Do not uninstall the original IVI driver when you create a MATLAB IVI instrument driver based on an IVI driver. You need the IVI driver in order to use the new MATLAB IVI instrument driver.

Constructing Device Objects Using a MATLAB IVI Instrument Driver

Once you have the MATLAB IVI instrument driver, create the device object with the file name of the MATLAB IVI instrument driver as an argument for `icdevice`. The following examples show the creation of the MATLAB IVI instrument driver (all with `.mdd` extensions) and the construction of device objects to use them.

See the `icdevice` and `makemid` reference pages for full details on these functions.

In the following example, `makemid` uses a `LogicalNames` value to identify an IVI driver, then creates a MATLAB IVI instrument driver. Because `LogicalNames` is associated with a driver session and hardware asset, you do not need to pass a `RsrcName` to `icdevice` when constructing the device object.

```
makemid('MainScope','MainScope.mdd');
obj = icdevice('MainScope.mdd');
```

In the next example, `makemid` uses a `ProgramID` to reference an IVI-COM driver, then creates a MATLAB IVI instrument driver. The device object requires a `RsrcName` in addition to the file name of the MATLAB IVI instrument driver.

```
makemid('TekScope.TekScope','TekScopeML.mdd');
obj = icdevice('TekScopeML.mdd','GPIB0::13::INSTR');
```

In the next example, `makemid` uses a software `Module` to reference an IVI-C driver, then creates a MATLAB IVI instrument driver. The device object requires a `RsrcName` in addition to the file name of the MATLAB IVI instrument driver.

```
makemid('ag3325b','Ag3325bML.mdd');
obj = icdevice('Ag3325bML.mdd','ASRL1::INSTR');
```

In the next example, `makemid` creates a MATLAB IVI instrument driver based on the IVI-C class driver `ivifgen`. The device object uses the MATLAB IVI instrument driver file name and the logical name of the driver from the IVI configuration store.

```
makemid('ivifgen','FgenML.mdd');
obj = icdevice('FgenML.mdd','FuncGen');
```

Using MATLAB IVI Wrappers

MATLAB IVI wrappers work well with class-compliant drivers.

This example shows how to connect to an instrument and read a waveform using a MATLAB IVI Wrapper.

The instrument in this example is an Agilent Technologies' MSO6014 mixed signal oscilloscope, with an Agilent546XX driver.

```
%Create the object
myScope = instrument.ivicom.IviScope('Agilent546XX.Agilent546XX');

%Connect to the instrument using the VISA resource string
myScope.Initialize('TCPIP0::xxx-xxxx.xxx.<yourdomain.com>::inst0::INSTR',false,
false,'simulate=false');
```

```
%Access the Measeurements Collection
myScopeMeasurements  = myScope.Measurements

%Configure measurement 1
myScopeMeasurements.AutoSetup;
name = myScopeMeasurements.Name(1);
myScopeMeasurement1 = myScopeMeasurements.Item(name);

%Access the Channels collection
myScopeChannels = myScope.Channels;

%Configure channel 1
name = myScopeChannels.Name(1);
myScopeChannel1=  myScopeChannels.Item(name)
myScopeChannel1.Enabled = 1;

%Configure a trigger
myScope.Trigger.Source = 'Channel1';
myScope.Trigger.Level = 1.O;
myScope.Trigger.Edge.Slope = 'IviScopeTriggerSlopePositive';

%Start the measurement and get the data
myScopeMeasurements.Initiate;
myWaveform = myScopeMeasurement1.FetchWaveform;

%Plot the data
plot(myWaveform);

%Close and delete the object
myScope.Close;
myScope.delete
```
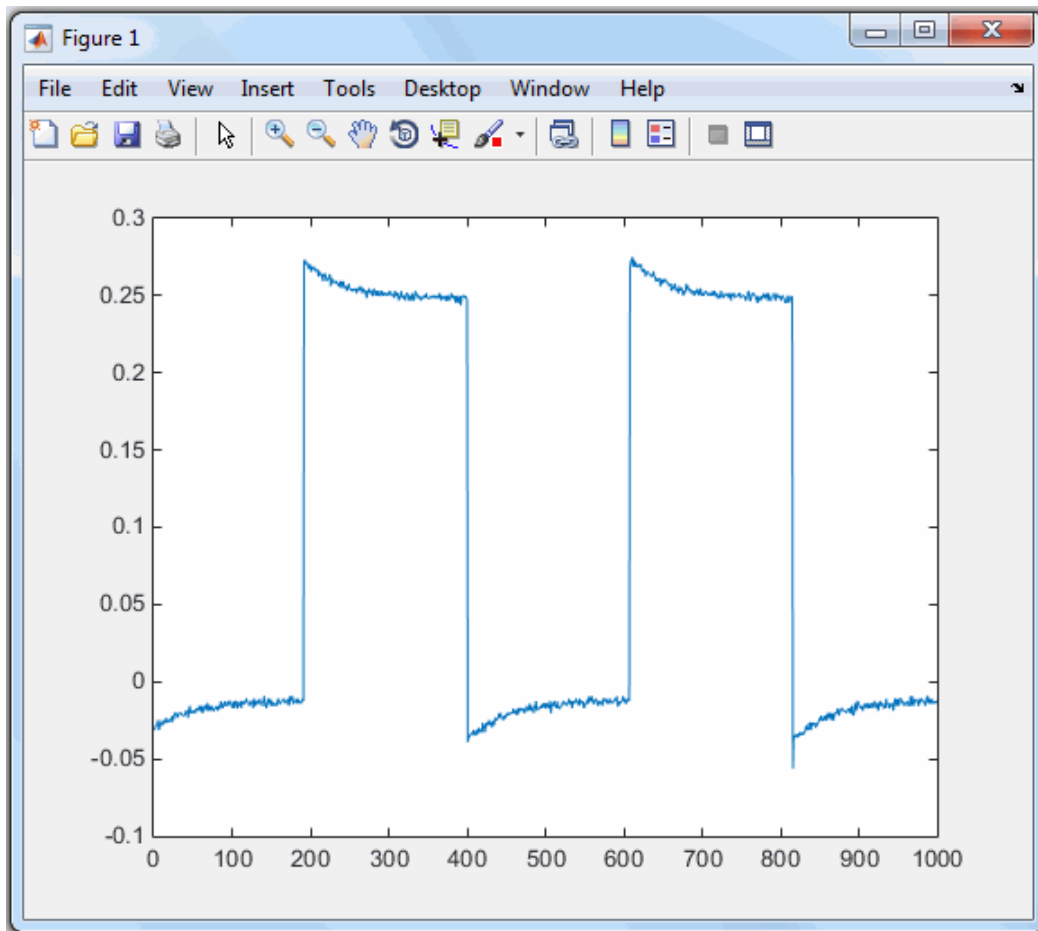
Plot the Waveform Read Using the MATLAB IVI Wrapper

IVI Configuration Store

In this section...
"Benefits of an IVI Configuration Store" on page 13-16
"Components of an IVI Configuration Store" on page 13-16
"Configuring an IVI Configuration Store" on page 13-17

Benefits of an IVI Configuration Store

By providing a way to configure the relationship between drivers and I/O references, an IVI configuration store greatly enhances instrument interchangeability.

Suppose your code uses only a specified driver to communicate with one type of instrument at a fixed location. If you change the instrument model, instrument location, or driver, you would have to modify the code to accommodate that change.

An IVI configuration store offers the ability to accommodate different instrument models, drivers, or ports, without having to modify your code. This interchangeability is especially useful when you use code that cannot be easily modified.

Components of an IVI Configuration Store

The components of an IVI configuration store identify:

- Locations of the instruments to communicate with
- Software modules used to control the instruments
- Associations of software modules used with instruments at specific locations

Component	Description
Software module	A software module is instrument-specific, and contains the commands and functions necessary to communicate with the instrument. The instrument vendor commonly provides software modules, which you cannot edit from the MATLAB Command Window.
Hardware asset	A hardware asset identifies a communication port connected the instrument. Configure this component with an `IOResourceDescriptor`. Usually you have one hardware asset per connection location (protocol type, bus address, and so on).
Driver session	A driver session makes the association between a software module and a hardware asset. Generally, you have a driver session for each instrument at each of its possible locations. Identical instruments connected at different locations can use the same software module, but because they have different hardware assets, they require different driver sessions. Different kinds of instruments connected to the same location (at different times) use the same hardware asset, but can have different software modules. Therefore, they require different driver sessions.
Logical name	A logical name is a configuration store component that provides access to a driver session. You can interpret a logical name as a configurable pointer to a driver session. In a typical setup, the code communicates with an instrument via a logical name. If the code must communicate with a different instrument (for example, a similar scope at a different location), update only the logical name within the IVI configuration store to point to the new driver session. You need not rewrite any code because it uses the same logical name.

Configuring an IVI Configuration Store

Using the GUI

You can use the Test & Measurement Tool to examine or configure your IVI configuration store. Open the tool by typing:

```
tmtool
```

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

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

Using the Command Line

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

```
instrhwinfo('ivi')
ans =
            LogicalNames: {'MainScope', 'FuncGen'}
```

```
               ProgramIDs: {'TekScope.TekScope','Agilent33250'}
                   Modules: {'ag3325b', 'hpe363xa'}
ConfigurationServerVersion: '1.6.0.10124'
  MasterConfigurationStore: 'C:\Program Files\IVI\Data\
                             IviConfigurationStore.xml'
                IVIRootPath: 'C:\Program Files\IVI\'
```

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

```
instrhwinfo('ivi','MainScope')
ans =
               DriverSession: 'TekScope.DriverSession'
               HardwareAsset: 'TekScope.Hardware'
              SoftwareModule: 'TekScope.Software'
        IOResourceDescriptor: 'GPIB0::13::INSTR'
  SupportedInstrumentModels: 'TekScope 5000, 6000 and 7000 series'
           ModuleDescription: 'TekScope software module desc'
              ModuleLocation: ''
```

You can use the command line to change the configuration store. Here is an example of changing it programmatically.

```
% Construct a configStore.
configStore = iviconfigurationstore;

% Set up the hardware asset with name myScopeHWAsset, and resource descriptor
%    TCPIP0::a-m6104a-004598::INSTR.
add(configStore, 'HardwareAsset', 'myScopeHWAsset', 'TCPIP0::a-m6104a-004598::INSTR');

% Add a driver session with name myScopeSession, and use the asset created in the step above.
%    Ag546XX is the Agilent driver.
add(configStore, 'DriverSession', 'myScopeSession', 'Ag546XX', 'myScopeHWAsset');

% Add a logical name to the configStore, with name myScope and driver session
%    named myScopeSession.
add(configStore, 'LogicalName', 'myScope', 'myScopeSession');

% Save the changes to the IVI configuration store data file.
commit(configStore);

% You can verify that the steps you just performed worked.
logicalNameInfo = instrhwinfo('ivi', 'myscope')
```

Basic IVI Configuration Store

Following is an example of configuration used by `data_analyzer.m`.
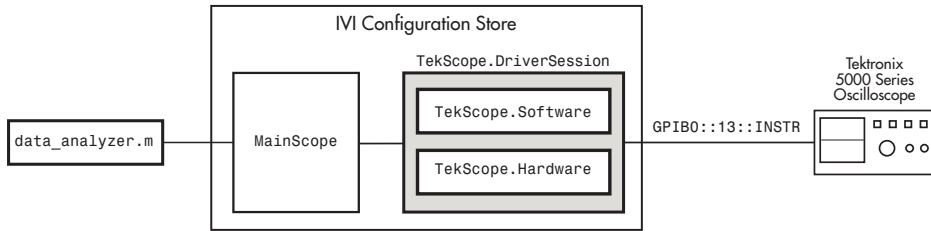
Create and configure elements in the IVI configuration store using the IVI configuration store object methods `add`, `commit`, `remove`, and `update`. For further details, see the reference pages for these methods.

IVI Configuration Store with Several Interchangeable Elements

The following figure shows an example of an IVI configuration store with several interchangeable components. **Code 1** requires access to the oscilloscopes at two different locations (hardware asset X and hardware asset Y). The scopes are similar, so they use the same software module S. Here, the scopes are at different locations (or the same scope connected to two different locations at different times). Therefore, each configuration requires its own driver session, in this example, driver session A and driver session B.
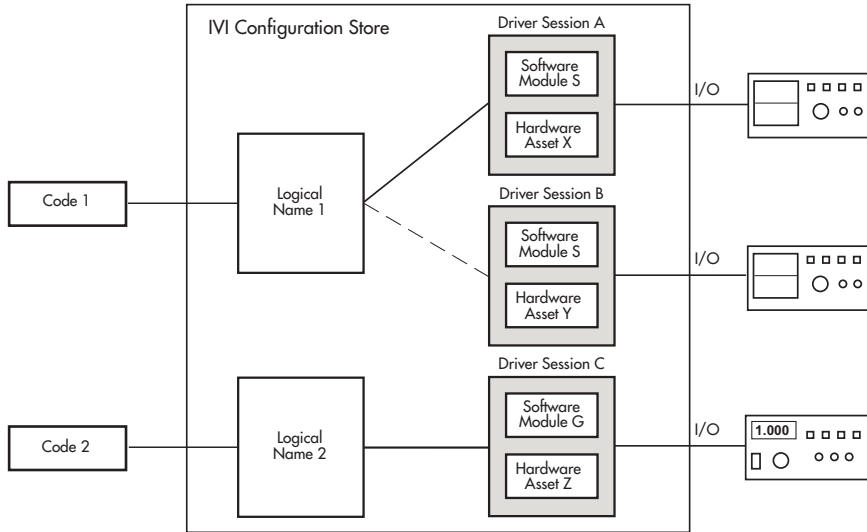
Write **Code 1** to access logical name 1. You configure the name in the IVI configuration store to access driver session A or driver session B (but not both at the same time). Because you select the driver session in the IVI configuration store, you need not alter the code to change access from one scope to the other.

Using IVI-C Class-Compliant Wrappers

In this section...
"IVI-C Wrappers" on page 13-22
"Prerequisites" on page 13-22
"Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI" on page 13-23
"Reading Waveforms Using the IVI-C Class Compliant Interface" on page 13-23
"IVI-C Class Compliant Wrappers in Test & Measurement Tool" on page 13-24

IVI-C Wrappers

The IVI-C wrappers provide an interface to MATLAB for instruments running on IVI-C class-compliant drivers.

This documentation example uses a specific instrument, an Agilent MSO6104A oscilloscope. This feature works with any IVI-C class compliant instrument. You can follow the basic steps, using your particular instrument if the device is IVI-C class compliant.

Prerequisites

To use the wrapper you must have the following software installed.

- Windows 32-bit or Windows 64-bit
- VISA shared components
- VISA

    The following example uses Agilent VISA, but you can use any version of VISA.
- National Instruments compliance package NICP 4.1
- Your instrument driver

You can use `instrhwinfo` to confirm that these software modules are installed.

```
% Check that the software is properly installed.
instrhwinfo('ivi')
```

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

When using IVI-C or VXI Plug&Play drivers, executing your code will generate additional file(s) in the folder specified by executing the following code at the MATLAB prompt:

```
fullfile(tempdir,'ICTDeploymentFiles',sprintf('R%s',version('-release')))
```

On all supported platforms, a file with the name MATLABPrototypeFor<driverName>.m is generated, where <driverName> the name of the IVI-C or VXI Plug&Play driver. With 64-bit MATLAB on Windows, a second file by the name <driverName>_thunk_pcwin64.dll is generated. When creating your deployed application or shared library, manually include these generated files. For more information on including additional files refer to the MATLAB Compiler documentation.

Reading Waveforms Using the IVI-C Class Compliant Interface

This example shows the general workflow to use with an IVI-C class-compliant device. This example uses a specific instrument, an Agilent MSO6104A oscilloscope. This feature works with any IVI-C class-compliant instrument. You can follow the basic steps using your particular instrument if it is IVI-C class-compliant.

1   Ensure all necessary software is installed. See "Prerequisites" on page 13-22 for the list.

2   Ensure that your instrument is recognized by the VISA utility. In this case, open Agilent Connectivity Expert and make sure it recognizes the oscilloscope.

3   Set up the logical name using the Configuration Store. The VISA resource string shown in this code was acquired from the VISA utility in step 2.

```
% Construct a configStore.
configStore = iviconfigurationstore;

% Set up the hardware asset called myScopeHWAsset, and resource description
    TCPIP0::a-m6104a-004598::INSTR.
add(configStore, 'HardwareAsset', 'myScopeHWAsset', 'TCPIP0::a-m6104a-004598::INSTR');

% Add a driver session called myScopeSession, and use the asset created in the
    step above. Ag546XX is the Agilent driver version.
add(configStore, 'DriverSession', 'myScopeSession', 'Ag546XX', 'myScopeHWAsset');

% Add a logical name to the configStore called myScope and driver session called
    myScopeSession.
```

```
add(configStore, 'LogicalName', 'myScope', 'myScopeSession');

% Save the changes to the IVI configuration store data file.
commit(configStore);

% You can verify that the steps you just performed worked.
logicalNameInfo = instrhwinfo('ivi', 'myscope')
```

For more information about the configuration store, see "IVI Configuration Store" on page 13-16.

4 Create an instance of the scope.

```
% Instantiate an instance of the scope.
ivicScope = instrument.ivic.IviScope();
```

5 Connect to the instrument.

```
% Open the hardware session.
ivicScope.init('myScope', true, true);
```

6 Communicate with the instrument. For example, read a waveform.

```
% Use the AutoSetup method to automatically set up the oscilloscope.
ivicScope.Configuration.AutoSetup();

% Create a record length variable.
recordLength = ivicScope.Acquisition.Horizontal_Record_Length;

% Preallocate buffer to store the data read from the scope.
waveformArray = zeros(1, recordLength);

% Read a waveform with channel name set to channel1 and timeout to 1000.
[waveformArray,actualPoints,initiaX,xIncrement] = ivicScope.WaveformAcquisition.
    ReadWaveform('channel1', recordLength, 1000, waveformArray);

% Plot the waveform and assign labels for the plot.
plot(waveformArray);
xlabel('Samples');
ylabel('Voltage');
```

7 After configuring the instrument and retrieving its data, close the session and remove it from the workspace.

```
ivicScope.close();
clear ivicScope;
```

IVI-C Class Compliant Wrappers in Test & Measurement Tool

You can also use the IVI-C Wrappers functionality from the Test & Measurement Tool. View the IVI-C nodes by setting a preference in MATLAB.

1   In MATLAB, on the **Home** tab, in the **Environment** section, click **Preferences**. Then select **Instrument Control** in the Preferences dialog box.

2   Select the **Show IVI Instruments in TMTool** option in the **IVI Instruments** section.

   If you do not have the required software installed, you will get a message indicating that. See "Prerequisites" on page 13-22 for the list of required software.

3   Start the Test & Measurement Tool (using the `tmtool` function), and the new **IVI Instruments** node appears under **Instrument Drivers**.

For information on using it in the Test & Measurement Tool, see the Help within the tool by selecting the **IVI Instruments** node in the tree once it is visible after setting the MATLAB preference.

Using Quick-Control Oscilloscope

In this section...

Quick-Control Oscilloscope

You can use the Quick-Control Oscilloscope for any oscilloscope that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. You can also use it for Tektronix oscilloscopes. This oscilloscope object is an easy to use.

This documentation example uses a specific instrument, an Agilent MSO6104 oscilloscope. This feature works with any IVI-C class oscilloscope. You can follow the basic steps, using your particular instrument.

Quick-Control Oscilloscope Prerequisites

Using IVI-C

To use the Quick-Control Oscilloscope for an IVI-C scope, you must have the following software installed.

- Windows 32-bit or Windows 64-bit platforms
- VISA shared components
- VISA

  Note, the following example uses Agilent VISA, but you can use any version of VISA.
- National Instruments IVI compliance package NICP 4.1
- Your instrument's device-specific driver

You can use `instrhwinfo` to confirm that the required software is installed.

```
% Check that the software is properly installed.
instrhwinfo('ivi')
```

Reading Waveforms Using the Quick-Control Oscilloscope

This example shows the general workflow to use for the Quick-Control Oscilloscope. This example uses a specific instrument, an Agilent MSO6104 oscilloscope. This feature works with any oscilloscope using an IVI-C driver. You can follow the basic steps using your particular scope. For use with a Tektronix scope, see the example in the next section.

1 Ensure all necessary software is installed. See "Quick-Control Oscilloscope Prerequisites" on page 13-26 for the list.

2 Ensure that your instrument is recognized by the VISA utility. In this case, open Agilent Connectivity Expert and make sure it recognizes the oscilloscope.

3 Create an instance of the oscilloscope.

```
% Instantiate an instance of the scope.
myScope = oscilloscope()
```

4 Discover available resources. A resource string is an identifier to the instrument. You must set it before connecting to the instrument.

```
% Find resources.
availableResources = getResources(myScope)
```

This returns a resource string or an array of resource strings.

```
availableResources =
   TCPIP0::a-m6104a-004598.dhcp.mathworks.com::inst0::INSTR
```

5 Connect to the scope.

If multiple resources are available, use the VISA utility to verify the correct resource and set it.

```
myScope.Resource = 'TCPIP0::a-m6104a-004598::inst0::INSTR';

% Connect to the scope.
connect(myScope);
```

6 Configure the oscilloscope.

You can configure any of the scope's properties that are able to be set. In this example enable channel 1 and configure various acquisition settings as shown.

```
% Automatically configure the scope based on the input signal.
autoSetup(myScope);

% Set the acquisition time to 0.01 second.
myScope.AcquisitionTime = 0.01;

% Set the acquisition to collect 2000 data points.
myScope.WaveformLength = 2000;
```

7 Communicate with the instrument. For example, read a waveform.

In this example, the `getWaveform` function returns the waveform that was acquired using the front panel of the scope. The function can also initiate an acquisition on the enabled channel and then return the waveform after the acquisition. For examples on all the use cases for this function, see `getWaveform`.

```
% Acquire the waveform.
waveformArray = getWaveform(myScope);

% Plot the waveform and assign labels for the plot.
plot(waveformArray);
xlabel('Samples');
ylabel('Voltage');
```

8 After configuring the instrument and retrieving its data, close the session and remove it from the workspace.

```
disconnect(myScope);
clear myScope;
```

Reading a Waveform Using a Tektronix Scope

Reading a waveform with a Tektronix scope using Quick-Control Oscilloscope is basically the same workflow as described in the previous example using an Agilent scope with VISA. But the resource and driver information is different.

If you use the `getResources` function, instead of getting a VISA resource string as shown in step 4 of the previous example, you will get the interface resource of the Tektronix scope. For example:

```
% Find resources.
availableResources = getResources(myScope)
```

This returns the interface resource information.

```
availableResources =
   GPIBO::AGILENT::7::1O
```

Where `gpib` is the interface being used, `agilent` is the interface type for the adaptor that the Tektronix scope is connected to, and the numbers are interface constructor parameters.

If you use the `getDrivers` function, you get information about the driver and its supported instrument models. For example:

```
% Get driver information.
drivers = getDrivers(myScope)
```

This returns the driver and instrument model information.

```
Driver: tekronix
Supported Models:
   TDS200, TDS1000, TDS2000, TDS1000B, TDS2000B, TPS2000
   TDS3000, TDS3000B, MSO4000, DPO4000, DPO7000, DPO7000B
```

This example shows the general workflow to use for the Quick-Control Oscilloscope for a Tektronix scope. This feature works with any supported oscilloscope model. You can follow the basic steps using your particular scope.

1   Create an instance of the oscilloscope.

```
% Instantiate an instance of the scope.
myScope = oscilloscope()
```

2   Discover available resources. A resource string is an identifier to the instrument. You must set it before connecting to the instrument.

```
% Find resources.
availableResources = getResources(myScope)
```

This returns a resource string or an array of resource strings.

```
availableResources =
  GPIB0::AGILENT::7::10
```

Where gpib is the interface being used, agilent is the interface type for the adaptor that the Tektronix scope is connected to, and the numbers are interface constructor parameters.

3   Connect to the scope.

```
% Connect to the scope.
connect(myScope);
```

4   Configure the oscilloscope.

You can configure any of the scope's properties that are able to be set. In this example enable channel 1 and set acquisition time as shown. You can see examples of other acquisition parameters in step 6 of the previous example.

```
% Set the acquisition time to 0.01 second.
myScope.AcquisitionTime = 0.01;

% Set the acquisition to collect 2000 data points.
set(myScope, 'WaveformLength', 2000);

% Enable channel 1.
enableChannel(myScope, 'CH1');
```

5   Communicate with the instrument. For example, read a waveform.

In this example, the getWaveform function returns the waveform that was acquired using the front panel of the scope. The function can also initiate an acquisition on the enabled channel and then return the waveform after the acquisition. For examples on all the use cases for this function, see getWaveform.

```
% Acquire the waveform.
```

```
waveformArray = getWaveform(myScope);

% Plot the waveform and assign labels for the plot.
plot(waveformArray);
xlabel('Samples');
ylabel('Voltage');
```

6 After configuring the instrument and retrieving its data, close the session and remove it from the workspace.

```
disconnect(myScope);
clear myScope;
```

Quick-Control Oscilloscope Functions

The oscilloscope function can use the following special functions, in addition to standard functions such as connect and disconnect.

Function	Description
autoSetup	Automatically configures the instrument based on the input signal. `autoSetup(myScope);`

disableChannel	Disables the oscilloscope's channel(s).
	
	`disableChannel(myScope, 'Channel1');`
	`disableChannel(myScope, {'Channel1', 'Channel2'});`

enableChannel	Enables the oscilloscope's channel(s) from which waveform(s) will be retrieved.
	
	`enableChannel(myScope, 'Channel1');`
	`enableChannel(myScope, {'Channel1', 'Channel2'});`

getDrivers	Returns a list of available drivers with their supported instrument models. `drivers = getDrivers(myScope);`
getResources	Retrieves a list of available resources of instruments. It returns a list of available VISA resource strings when using an IVI-C scope. It returns the interface resource information when using a Tektronix scope. `res = getResources(myScope);`

13-31

Function	Description
`getVerticalCoupling`	Returns the value of how the oscilloscope couples the input signal for the selected channel name as a MATLAB string. Possible values returned are `'AC'`, `'DC'`, and `'GND'`. `VC = getVerticalCoupling (myScope, 'Channel1');`
`getVerticalOffset`	Returns the location of the center of the range for the selected channel name as a MATLAB string. The units are volts. `VO = getVerticalOffset (myScope, 'Channel1');`
`getVerticalRange`	Returns the absolute value of the input range the oscilloscope can acquire for selected channel name as a MATLAB string. The units are volts. `VR = getVerticalRange (myScope, 'Channel1');`
`getWaveform`	Returns the waveform(s) displayed on the scope screen. Retrieves the waveform(s) from enabled channel(s). `w = getWaveform(myScope);`
`reset`	Resets the device. `reset(myScope);`
`setVerticalCoupling`	Specifies how the oscilloscope couples the input signal for the selected channel name as a MATLAB string. Values are `'AC'`, `'DC'`, and `'GND'`. `setVerticalCoupling (myScope, 'Channel1', 'AC');`
`setVerticalOffset`	Specifies the location of the center of the range for the selected channel name as a MATLAB string. For example, to acquire a sine wave that spans from 0.0 to 10.0 volts, set this attribute to 5.0 volts. `setVerticalOffset (myScope, 'Channel1', 5);`
`setVerticalRange`	Specifies the absolute value of the input range the oscilloscope can acquire for the selected channel name as a MATLAB string. The units are volts. `setVerticalRange (myScope, 'Channel1', 10);`

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

When using IVI-C or VXI Plug&Play drivers, executing your code will generate additional file(s) in the folder specified by executing the following code at the MATLAB prompt:

```
fullfile(tempdir,'ICTDeploymentFiles',sprintf('R%s',version('-release')))
```

On all supported platforms, a file with the name MATLABPrototypeFor<driverName>.m is generated, where <driverName> the name of the IVI-C or VXI Plug&Play driver. With 64-bit MATLAB on Windows, a second file by the name <driverName>_thunk_pcwin64.dll is generated. When creating your deployed application or shared library, manually include these generated files. For more information on including additional files refer to the MATLAB Compiler documentation.

Using Quick-Control Function Generator

In this section...
"Quick-Control Function Generator" on page 13-34
"Quick-Control Function Generator Prerequisites" on page 13-34
"Generating Waveforms Using the Quick-Control Function Generator" on page 13-35
"Quick-Control Function Generator Functions" on page 13-38
"Quick-Control Function Generator Properties" on page 13-41
"Creating Shared Libraries or Standalone Applications When Using IVI-C or VXI" on page 13-44

Quick-Control Function Generator

A new way to communicate with instruments, Quick-Control Instruments, was introduced in R2011b with the Quick-Control Oscilloscope. In R2012a, a second instrument class is introduced – the Quick-Control Function Generator. You can use this new function generator, or fgen, for simplified fgen control and waveform generation.

Create the Quick-Control Function Generator object using the Instrument Control Toolbox `fgen` function. It simplifies controlling function generators and performs arbitrary waveform generations without dealing with the underlying drivers.

You can use the Quick-Control Function Generator for any function generator that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. This fgen object is easy to use.

The documentation examples use a specific instrument, a Tektronix AFG 3022B function generator. This feature works with any instrument that has IVI-C fgen class drivers. You can follow the basic steps, using your particular instrument.

Quick-Control Function Generator Prerequisites

To use the Quick-Control Function Generator for an IVI-C fgen, ensure the following software is installed:

- Windows 32-bit or Windows 64-bit platforms
- VISA shared components

- VISA

  Note, the following examples use Agilent VISA, but you can use any vendor's implementation of VISA.

- National Instruments IVI compliance package NICP 4.1 or 4.3
- Your instrument's device-specific driver

You can use `instrhwinfo` to confirm that the required software is installed.

```
% Check that the software is properly installed.
instrhwinfo('ivi')
```

Generating Waveforms Using the Quick-Control Function Generator

The first example shows the general workflow to use for the Quick-Control Function Generator for a standard waveform. The second example shows the general workflow to use for the Quick-Control Function Generator for an arbitrary waveform. These examples use a specific instrument, but they work with any function generator using an IVI-C driver as long as the instrument and the driver support the functionality. You can follow the basic steps using your particular function generator.

Generate Standard Waveform

In this example, an electronic test engineer wants to create a simple sine waveform to test the clock operating range of a digital circuit.

1 Ensure all necessary software is installed. See "Quick-Control Function Generator Prerequisites" on page 13-34 for the list.

2 Create an instance of the function generator.

```
% Instantiate an instance of the fgen.
myFGen = fgen();
```

3 Discover available resources. A resource string is an identifier to the instrument. You must set it before connecting to the instrument.

```
% Find resources.
availableResources = getResources(myFGen)
```

This returns a resource string or an array of resource strings, for example:

```
ans =
```

```
ASRL::COM1
GPIB0::INTFC
GPIB0::10::INSTR
PXI0::MEMACC
TCPIP0::172.28.16.153::inst0::INSTR
TCPIP0::172.28.16.174::inst0::INSTR
```

4 Set the resource for the function generator you want to communicate with.

```
myFGen.Resource = 'GPIB0::10::INSTR';
```

5 Connect to the function generator.

```
connect(myFGen);
```

6 Specify the channel name from which the function generator produces the waveform.

```
selectChannel(myFGen, '1');
```

7 Configure the function generator.

You can configure any of the instrument's properties that are settable. Configure the waveform to be a continuous sine wave and then configure various settings as shown.

```
% Set the type of waveform to a sine wave.
myFGen.Waveform = 'sine';

% Set the output mode to continuous.
myFGen.Mode = 'continuous';

% Set the load impedance to 50 Ohms.
myFGen.OutputImpedance = 50;

% Set the frequency to 2500 Hz.
myFGen.Frequency = 2500;

% Set the amplitude to 1.2 volts.
myFGen.Amplitude = 1.2;

% Set the offset to 0.4 volts.
myFGen.Offset = 0.4;
```

8 Enable signal generation with the instrument, for example, output signals.

In this example, the `enableOutput` function enables the function generator to produce a signal that appears at the output connector.

```
% Enable the output of signals.
enableOutput(myFGen);
```

When you are done, disable the output.

```
% Disable the output of signals.
disableOutput(myFGen);
```

9 After configuring the instrument and generating a signal, close the session and remove it from the workspace.

```
disconnect(myFGen);
clear myFgen;
```

Generate Arbitrary Waveform

In this example, an electronic design engineer wants to generate a complex waveform with MATLAB, then download them into the function/arbitrary waveform generator and output them one after the other, and then finally remove the downloaded waveforms afterward. In this example we are using the GPIB interface.

1 Ensure all necessary software is installed. See "Quick-Control Function Generator Prerequisites" on page 13-34 for the list.

2 Create an instance of the function generator.

```
% Instantiate an instance of the fgen.
myFGen = fgen();
```

3 Set the resource.

```
myFGen.Resource = 'GPIB0::10::INSTR';
```

4 Connect to the function generator.

```
connect(myFGen);
```

5 Specify the channel name from which the function generator produces the waveform.

```
selectChannel(myFGen, '1');
```

6 Configure the function generator.

You can configure any of the instrument's properties that are settable. Configure the waveform to be a continuous arbitrary wave.

```
% Set the type of waveform to an arbitrary wave.
myFGen.Waveform = 'arb';

% Set the output mode to continuous.
myFGen.Mode = 'continuous';
```

7 Communicate with the instrument.

In this example, create the waveform, then download it to the function generator using the downloadWaveform function. Then enable the output using the enableOutput function, and then remove the waveform using the removeWaveform function.

```
% Create the waveform.
w1 = 1:0.001:2;

% Download the waveform to the function generator.
h1 = downloadWaveform (myFGen, w1);

% Enable the output.
enableOutput(myFGen);
```

When you are done, remove the waveforms.

```
% Remove the waveform.
removeWaveform(myFGen);
```

8 After communicating with the instrument, close the session and remove it from the workspace.

```
disconnect(myFGen);
clear myFgen;
```

Quick-Control Function Generator Functions

The fgen function uses the following functions, in addition to standard functions such as connect and disconnect.

Function	Description
selectChannel	Specifies the channel name from which the function generator produces the waveform. Example: selectChannel(myFGen, '1');
getDrivers	Returns a list of available function generator instrument drivers. Example:

Function	Description
	`drivers = getDrivers(myFGen);` See the note following this table about using a SCPI-based driver for Agilent function generators.
getResources	Retrieves a list of available instrument resources. It returns a list of available VISA resource strings when using an IVI-C function generator. Example: `res = getResources(myFGen);`
selectWaveform	Specifies which arbitrary waveform the function generator produces. Example: `selectWaveform (myFGen, wh);` where wh is the waveform handle you are selecting.

downloadWaveform	Downloads an arbitrary waveform to the function generator. If you provide an output variable, a waveform handle is returned. It can be used in the `selectWaveform` and `removeWaveform` functions.
	
	If you don't provide an output variable, function generator overwrites the waveform when a new waveform is downloaded and deletes it upon disconnection.
	
	Example:
	
	`% Download the following waveform to fgen`
	`w = 1:0.001:2;`
	`downloadWaveform (myFGen, w);`
	
	`% Download a waveform to fgen and return`
	`    a waveform handle`
	`wh = downloadWaveform (myFGen, w);`

Function	Description

removeWaveform	Removes a previously created arbitrary waveform from the function generator's memory. If a waveform handle is provided, it removes the waveform represented by the waveform handle.
	
	Example:
	
	``
	% Remove a waveform from fgen with waveform
	    handle 10000
	  removeWaveform (myFGen, 10000);
	``

enableOutput	Enables the function generator to produce a signal that appears at the output connector. This function produces a waveform defined by the Waveform property. If the Waveform property is set to 'Arb', the function uses the latest internal waveform handle to output the waveform. ``` enableOutput (myFGen); ```
disableOutput	Disables the signal that appears at the output connector. Disables the selected channel. ``` disableOutput (myFGen); ```
reset	Sets the function generator to factory state.

Using a SCPI-based Driver for Agilent Function Generators

If you are using a SCPI-based Agilent function generator such as the 33220A, you will see the following when you use the getDrivers function on an fgen object myFGen.

```
drivers = getDrivers(myFGen);

drivers =

Driver: Agilent332x0_SCPI
Supported Models:
   33210A, 33220A, 33250A
```

The _SCPI after the instrument name indicates this is using a SCPI driver instead of the IVI driver.

Quick-Control Function Generator Properties

The fgen function can use the following properties.

Property	Description
AMDepth	Specifies the extent of Amplitude modulation the function generator applies to the carrier signal. The units are a percentage of full modulation. At 0% depth, the output amplitude equals the carrier signal's amplitude. At 100% depth, the output amplitude equals twice the carrier signal's amplitude. This property affects function generator behavior only when the Mode is set to 'AM' and ModulationResource is set to 'internal'.
Amplitude	Specifies the amplitude of the standard waveform. The value is the amplitude at the output terminal. The units are volts peak-to-peak (Vpp). For example, to produce a waveform ranging from -5.0 to +5.0 volts, set this value to 10.0 volts. Does not apply if Waveform is of type 'Arb'.
ArbWaveformGain	Specifies the factor by which the function generator scales the arbitrary waveform data. Use this property to scale the arbitrary waveform to ranges other than -1.0 to +1.0. When set to 2.0, the output signal ranges from -2.0 to +2.0 volts. Only applies if Waveform is of type 'Arb'.
BurstCount	Specifies the number of waveform cycles that the function generator produces after it receives a trigger. Only applies if Mode is set to 'burst'.
ChannelNames	This read-only property provides available channel names in a cell array.
Driver	This property is optional. Use only if necessary to specify the underlying driver used to communicate with an instrument. If the DriverDetectionMode property is set to 'manual', use the Driver property to specify the instrument driver.
DriverDetectionMode	Sets up criteria for connection. Valid values are 'auto' and 'manual'. The default value is 'auto', which means you do not need to set a driver name before connecting to an instrument. If set to 'manual', a driver name needs to

Property	Description
	be provided using the `Driver` property before connecting to instrument.
FMDeviation	Specifies the maximum frequency deviation the modulating waveform applies to the carrier waveform. This deviation corresponds to the maximum amplitude level of the modulating signal. The units are Hertz (Hz). This property affects function generator behavior only when `Mode` is set to `'FM'` and `ModulationSource` is set to `'internal'`.
Frequency	Specifies the rate at which the function generator outputs an entire arbitrary waveform when Waveform is set to `'Arb'`. It specifies the frequency of the standard waveform when Waveform is set to standard waveform types. The units are Hertz (Hz).
Mode	Specifies run mode. Valid values are `'continuous'`, `'burst'`, `'AM'`, or `'FM'`. Specifies how the function generator produces waveforms. It configures the instrument to generate output continuously or to generate a discrete number of waveform cycles based on a trigger event. It can also be set to AM and FM.
ModulationFrequency	Specifies the frequency of the standard waveform that the function generator uses to modulate the output signal. The units are Hertz (Hz). This attribute affects function generator behavior only when `Mode` is set to `'AM'` or `'FM'` and the `ModulationSource` attribute is set to `'internal'`.
ModulationSource	Specifies the signal that the function generator uses to modulate the output signal. Valid values are `'internal'` and `'external'`. This attribute affects function generator behavior only when `Mode` is set to `'AM'` or `'FM'`.
ModulationWaveform	Specifies the standard waveform type that the function generator uses to modulate the output signal. This affects function generator behavior only when `Mode` is set to `'AM'` or `'FM'` and the `ModulationSource` is set to `'internal'`. Valid values are `'sine'`, `'square'`, `'triangle'`, `'RampUp'`, `'RampDown'`, and `'DC'`.

Property	Description
Offset	Uses the standard waveform DC offset as input arguments if the waveform is not of type `'Arb'`. Use Arb Waveform Offset as input arguments if the waveform is of type `'Arb'`. Specifies the DC offset of the standard waveform when Waveform is set to standard waveform. For example, a standard waveform ranging from +5.0 volts to 0.0 volts has a DC offset of 2.5 volts. When Waveform is set to `'Arb'`, this property shifts the arbitrary waveform's range. For example, when it is set to 1.0, the output signal ranges from 2.0 volts to 0.0 volts.
OutputImpedance	Specifies the function generator's output impedance at the output connector.
Resource	Set this before connecting to the instrument. It is the VISA resource string for your instrument.
SelectedChannel	Returns the selected channel name that was set using the `selectChannel` function.
StartPhase	Specifies the horizontal offset in degrees of the standard waveform the function generator produces. The units are degrees of one waveform cycle. For example, a 180-degree phase offset means output generation begins halfway through the waveform.
Status	This read-only property indicates the communication status of your instrument session. It is either `'open'` or `'closed'`.
TriggerRate	Specifies the rate at which the function generator's internal trigger source produces a trigger, in triggers per second. This property affects function generator behavior only when the `TriggerSource` is set to `'internal'`. Only applies if Mode is set to `'burst'`.
TriggerSource	Specifies the trigger source. After the function generator receives a trigger, it generates an output signal if Mode is set to `'burst'`. Valid values are `'internal'` or `'external'`.

Property	Description
`Waveform`	Uses the waveform type as an input argument. Valid values are `'Arb'`, for an arbitrary waveform, or these standard waveform types – `'Sine'`, `'Square'`, `'Triangle'`, `'RampUp'`, `'RampDown'`, and `'DC'`.

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

When using IVI-C or VXI Plug&Play drivers, executing your code will generate additional file(s) in the folder specified by executing the following code at the MATLAB prompt:

```
fullfile(tempdir,'ICTDeploymentFiles',sprintf('R%s',version('-release')))
```

On all supported platforms, a file with the name `MATLABPrototypeFor<driverName>.m` is generated, where `<driverName>` the name of the IVI-C or VXI Plug&Play driver. With 64-bit MATLAB on Windows, a second file by the name `<driverName>_thunk_pcwin64.dll` is generated. When creating your deployed application or shared library, manually include these generated files. For more information on including additional files refer to the MATLAB Compiler documentation.

14

Instrument Support Packages

Install the Ocean Optics Spectrometers Support Package

You can use Instrument Control Toolbox to communicate with Ocean Optics USB spectrometers. You can acquire data from the spectrometer and control it. Ocean Optics manufactures a broad line of USB-powered spectrometers covering the visible, near IR, and UV portions of the spectrum. You can use these spectrometers from MATLAB on Windows and Linux platforms.

The Instrument Control Toolbox Support Package for Ocean Optics Spectrometers lets you use MATLAB for comprehensive control of any spectrometer that is supported by the Ocean Optics OmniDriver software (version 2.12 or higher). You can perform many tasks, including:

- Acquire a spectrum
- Set the integration time
- Enable dark current and nonlinear spectral corrections
- View all connected devices

For a list of supported devices, see http://www.mathworks.com/hardware-support/ocean-optics-spectrometers.html.

This feature is available through the Hardware Support Packages. Using this installation process, download and install the following file(s) on your host computer:

- MATLAB Instrument Driver for Ocean Optics support
- Ocean Optics OmniDriver version 2.2 driver files
- An example that shows how to take measurements with an Ocean Optics spectrometer

Note: You can use this support package on a host computer running on 32-bit or 64-bit Windows or 64-bit Linux operating systems that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for Ocean Optics Spectrometers, in MATLAB type:

```
supportPackageInstaller
```

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select `Ocean Optics Spectrometers` from the list.

For more information about installing support packages, see "Supported Hardware".

Install the NI-SCOPE Support Package

You can use Instrument Control Toolbox to communicate with NI-SCOPE oscilloscopes. You can acquire waveform data from the oscilloscope and control it.

This feature is available through the Hardware Support Packages. Using this installation process, you download and install the following file(s) on your host computer:

- MATLAB Instrument Driver for NI-SCOPE support
- National Instruments driver file: NI-SCOPE driver version 3.9.7

Note: You can use this support package only on a host computer running a version of 32-bit or 64-bit Windows that Instrument Control Toolbox supports.

Supported Compiler Requirement

Certain functionality in the NI-SCOPE support package may require a supported compiler on your system. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks website.

Once you have installed a supported C compiler, set up your compiler by running `mex -setup`, as described in the documentation for `mex` in the MATLAB Function Reference.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-SCOPE Oscilloscopes, in MATLAB type:

```
supportPackageInstaller
```

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select `NI-SCOPE` from the list.

For more information about installing support packages, see "Supported Hardware".

Install the NI-FGEN Support Package

You can use the Instrument Control Toolbox to communicate with NI-FGEN function generators. You can control and configure the function generator, and perform tasks such as generating sine waves.

This feature is available through the Hardware Support Packages. Using this installation process, you download and install the following file(s) on your host computer:

- MATLAB Instrument Driver for NI-FGEN support
- National Instruments driver file: NI-FGEN driver version 2.9.1

Note: You can use this support package only on a host computer running a version of 32-bit or 64-bit Windows that Instrument Control Toolbox supports.

Supported Compiler Requirement

Certain functionality in the NI-FGEN support package may require a supported compiler on your system. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks website.

Once you have installed a supported C compiler, set up your compiler by running `mex -setup`, as described in the documentation for `mex` in the MATLAB Function Reference.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-FGEN Function Generators, in MATLAB type:

```
supportPackageInstaller
```

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select `NI-FGEN` from the list.

For more information about installing support packages, see "Supported Hardware".

Install the NI-DCPower Support Package

You can use Instrument Control Toolbox to communicate with NI-DCPower power supplies. You can control and take digital measurements from a power supply, such as the NI PXI 4011 triple-output programmable DC power supply.

To use this feature, download and install the following files on your host computer:

- MATLAB Instrument Driver for NI-DCPower support
- National Instruments NI-DCPower driver file
- Example that shows how to take digital measurements from an NI-DCPower power supply

Note: You can use this support package only on a host computer running a version of 32-bit or 64-bit Windows that Instrument Control Toolbox supports.

Supported Compiler Requirement

Certain functionality in the NI-DCPower support package may require a supported compiler on your system. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks website.

Once you have installed a supported C compiler, set up your compiler by running `mex -setup`, as described in the documentation for `mex` in the MATLAB Function Reference.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-DCPower Power Supplies, in MATLAB type:

`supportPackageInstaller`

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select `NI-DCPower` from the list.

For more information about installing support packages, see "Supported Hardware".

Install the NI-DMM Support Package

You can use Instrument Control Toolbox to communicate with NI-DMM digital multimeters. You can control and take measurements from a digital multimeter, such as measuring voltage or resistance.

To use this feature, download and install the following files on your host computer:

- MATLAB Instrument Driver for NI-DMM support
- National Instruments NI-DMM driver file
- Example that shows how to take digital measurements from a NI-DMM digital multimeter

Note: You can use this support package only on a host computer running a version of 32-bit or 64-bit Windows that Instrument Control Toolbox supports.

Supported Compiler Requirement

Certain functionality in the NI-DMM support package may require a supported compiler on your system. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks website.

Once you have installed a supported C compiler, set up your compiler by running `mex -setup`, as described in the documentation for `mex` in the MATLAB Function Reference.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-DMM Digital Multimeters, in MATLAB type:

`supportPackageInstaller`

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select `NI-DMM` from the list.

For more information about installing support packages, see "Supported Hardware".

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

For the Instrument Control Toolbox I2C and SPI interfaces, you can use either a Total Phase Aardvark host adaptor or an NI-845x adaptor. To use the I2C or SPI interface with the NI-845x adaptor, you must download this Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

To use the NI-845x driver, download and install the following files on your host computer:

- National Instruments NI-845x adaptor driver file
- Example that shows how to use the NI-854x adaptor with the I2C interface

---

Note: You can use this support package only on a host computer running a version of 32-bit or 64-bit Windows that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-845x I2C/SPI Interface, in MATLAB type:

```
supportPackageInstaller
```

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select `NI-845x I2C/SPI Interface` from the list.

For more information about installing support packages, see "Supported Hardware".

Install the Total Phase Aardvark I2C/SPI Support Package

For the Instrument Control Toolbox I2C and SPI interfaces, you can use either a Total Phase Aardvark host adaptor or an NI-845x adaptor. To use the I2C or SPI interface with the Aardvark adaptor, you must download this Hardware Support Package to obtain the driver, if you do not already have the driver installed. If you already have the driver installed, you do not need to download this support package.

The support package downloads and installs the Total Phase Aardvark host adaptor driver file on your host computer. Examples of using the Aardvark adaptor with the I2C interface can be found in the Instrument Control Toolbox documentation. For more information on using Aardvark with the I2C and SPI interfaces, see "I2C Interface Overview" on page 9-2 and "SPI Interface Overview" on page 10-2.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for Total Phase Aardvark I2C/SPI Interface, in MATLAB type:

```
supportPackageInstaller
```

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select `Total Phase Aardvark I2C/SPI Interface` from the list.

For more information about installing support packages, see "Supported Hardware".

Install the NI-Switch Support Package

You can use Instrument Control Toolbox to communicate with NI-Switch instruments. For example, you can control a relay box such as the NI PXI-2586 10-channel relay switch.

To use this feature, download and install the following files on your host computer:

· MATLAB Instrument Driver for NI-Switch support

· National Instruments NI-Switch driver file

· Example that shows how to control an NI-Switch relay switch

Note: You can use this support package only on a host computer running a version of 32-bit or 64-bit Windows that Instrument Control Toolbox supports.

Supported Compiler Requirement

Certain functionality in the NI-Switch support package may require a supported compiler on your system. For the current list of supported compilers, see Supported and Compatible Compilers on the MathWorks website.

Once you have installed a supported C compiler, set up your compiler by running `mex -setup`, as described in the documentation for `mex` in the MATLAB Function Reference.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for NI-Switch Hardware, in MATLAB type:

`supportPackageInstaller`

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select `NI-Switch` from the list.

For more information about installing support packages, see "Supported Hardware".

Install the National Instruments VISA and ICP Interfaces Support Package

The Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces lets you use the Quick Control Oscilloscope and Quick Control Function Generator interfaces.

After you download and install the support package, you can use the Quick Control interfaces to communicate with oscilloscopes and function generators.

The support package installs the following files on your host computer:

- MATLAB Instrument Driver for Quick Control Oscilloscope and Quick Control Function Generator
- VISA shared components
- VISA
- National Instruments IVI compliance package NICP 4.1 or later
- The Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces documentation

---

Note: You can use this support package only on a host computer running a version of 32-bit or 64-bit Windows that Instrument Control Toolbox supports.

Installing the Support Package

To install the Instrument Control Toolbox Support Package for National Instruments VISA and ICP Interfaces type:

supportPackageInstaller

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select NI-VISA and ICP from the list.

For more information about installing support packages, see "Supported Hardware".

Install the Keysight IO Libraries and VISA Interface Support Package

The Instrument Control Toolbox Support Package for Keysight™ IO Libraries and VISA Interface simplifies the use of Keysight (formerly Agilent) VISA by installing the necessary software components, such as the IO libraries and VISA shared components.

After you download and install the support package, you can use the VISA interface to communicate with Keysight instruments.

The support package installs the following files on your host computer:

- MATLAB Instrument Driver for Keysight VISA
- Keysight IO libraries
- Keysight VISA shared components

Installing the Support Package

To install the Instrument Control Toolbox Support Package for Keysight IO Libraries and VISA Interface type:

```
supportPackageInstaller
```

to open the Support Package Installer.

You can also open the installer from MATLAB by selecting **Home > Environment > Add-Ons > Get Hardware Support Packages**.

On the **Select support package to install** screen, select `Keysight IO Libraries` from the list.

For more information about installing support packages, see "Supported Hardware".

Using Generic Instrument Drivers

This chapter describes the use of generic drivers for controlling instruments from the MATLAB Command window, using the Instrument Control Toolbox software.

Generic Drivers: Overview

Generic drivers allow the Instrument Control Toolbox software to communicate with devices or software that do not use industry-standard drivers or protocols.

Typical cases, but not the only possibilities, are instruments that offer access through a COM interface (where the instrument can be accessed as an ActiveX® object from the MATLAB workspace), that use proprietary libraries, or that use custom MEX-files.

Because the generic nature of this feature does not lend itself to detailed discussion of specific instructions that work in all cases, the following sections of this chapter use an example to illustrate how to create and use a MATLAB generic instrument driver:

- "Writing a Generic Driver" on page 15-3
- "Using Generic Driver with Test & Measurement Tool" on page 15-8
- "Using a Generic Driver at Command Line" on page 15-11

Writing a Generic Driver

In this section...
"Creating the Driver and Defining Its Initialization Behavior" on page 15-3
"Defining Properties" on page 15-4
"Defining Functions" on page 15-7

Creating the Driver and Defining Its Initialization Behavior

In this example, the generic "instrument" that you control is Microsoft Internet Explorer® (IE), which is represented by a COM object. (This example works only on Windows systems.) Working through the example, you write a simple MATLAB instrument generic driver that allows the Instrument Control Toolbox software to communicate with a COM object. Using both a graphical interface and command-line code, with your driver you create an IE browser window, control its size, and specify what Web page it displays. The principles demonstrated in this example can be applied when writing a generic driver for any kind of instrument.

In this section, you create a new driver and specify what happens when an object is created for this driver.

1   Open the MATLAB Instrument Driver Editor from the MATLAB Command Window.

    midedit

2   To make it known that this driver is a generic driver, in the MATLAB Instrument Driver Editor, select **File > New > Generic driver**, as shown.
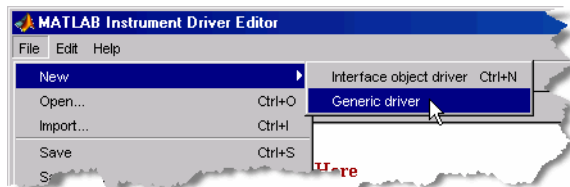
3   Select **File > Save as**.

    Navigate to the directory where you want to save your driver, and give it any name you want. This example uses the name ie_drv. Remember where you have saved your driver.

4   Select the Summary node in the driver editor window. Set the fields of this pane with any values you want. This example uses the following settings:

Manufacturer	Microsoft
Supported models	IE
Instrument type	Browser
Driver version	1.0

5   Select the node Initialization and Cleanup.
6   Click the **Create** tab.

This is where you define the code to execute when this driver is used to create a device object. This example identifies the COM object for Internet Explorer, and assigns the handle to that object as the Interface property of the device object being created.

7   Add the following lines of code to the **Create** tab:

```
ie = actxserver('internetexplorer.application');
obj.Interface = ie);
```

8   Click the **Connect** tab.

This is where you define the code to execute when you connect your device object to your instrument or software.

9   Add the following lines of code to the **Connect** tab:

```
ie = get(obj, 'Interface');
ie.Visible = 1);
ie.FullScreen = 0);
```

The first line gets ie as a handle to the COM object, based on the assignment in the **Create** code. The two lines after that set the window visibility and size.

Defining Properties

Writing properties for generic drivers in the MATLAB Instrument Driver Editor is a matter of writing straight code.

In this example, you define two properties. The first property uses the same name as the corresponding property of the COM object; the second property uses a different name from its corresponding COM object property.

Using the Same Name for a Property

The position of the IE browser window is determined by the `Top` and `Left` properties of its COM object. In the following steps, you make the `Top` property available to your device object through your generic driver. For this property, the name of the property is the same in both the COM object and in your device object.

1 Select the `Properties` node in the driver editor tree.
2 In the **Add property** field, enter the text `Top`, and click **Add**.
3 Expand the `Properties` node in the tree, and select the new node `Top`.
4 Click the **Property Values** tab. Your property can have a numeric value corresponding to screen pixels. For this example, you can limit the value of the property from `0` to `200`.
5 Make sure the **Data Type** field indicates `Double`. In the **Constraint** field, click the pull-down menu and select `Bounded`.
6 Keep the **Minimum** value of `0.0`, and enter a **Maximum** value of `200`.

Your driver editor window should look like the following figure.
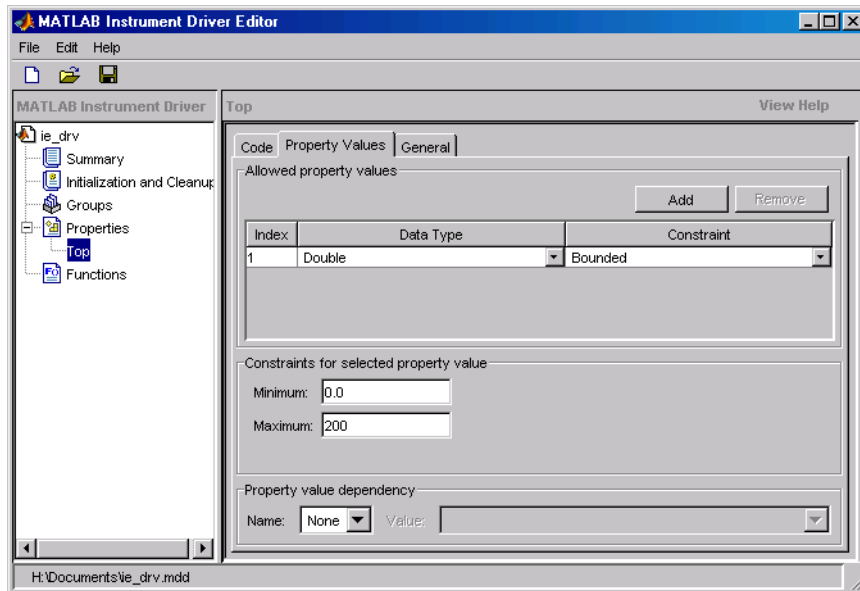
Now that you have defined the data type and acceptable values of the property, you can write the code to be executed whenever the device object property is accessed by `get` or `set`.

7 Click the **Code** tab.

The concept of reading the property is rather straightforward. When you get the `Top` property of the device object, the driver merely gets the value of the COM object's corresponding `Top` property. So all you need in the Get code function is to identify the COM object to get the information from.

8 Add the following code at the bottom of the function in the **Get code** pane:

```
ie = obj.Interface;
propertyValue = get(ie, propertyName);
```

The first line gets `ie` as a handle to the COM object. Remember that the `Interface` property of the device object is set to this value back in the driver's **Create** code. The second line retrieves the value of the COM object's `Top` property, and assigns it to `propertyValue`, which is returned to the `get` function for the device object.

9 Add the following code at the bottom of the function in the **Set code** pane:

```
ie = get(obj, 'Interface');
ie.propertyName = propertyValue;
```

Using a Different Name for a Property

In the preceding steps, you created in your driver a device object property that has the same name as the property of the COM object representing your instrument. You can also create properties with names that do not match those of the COM object properties. In the following steps, you create a property called `Vsize` that corresponds to the IE COM object property `Height`.

1 Select the `Properties` node in the driver editor tree.

2 In the **Add property** field, enter the text `Vsize`, and click **Add**.

3 Expand the `Properties` node in the tree, and select the new node `Vsize`.

4 Click the **Property Values** tab. This property can have a numeric value corresponding to screen pixels, whose range you define as `200` to `800`.

5 Make sure the **Data Type** field indicates `Double`. In the **Constraint** field, click the pull-down menu and select `Bounded`.

6 Enter a **Minimum** value of `200`, and enter a **Maximum** value of `800`.

7 Click the **Code** tab.

8 Add the following code at the bottom of the function in the **Get code** pane:

```
ie = obj.Interface;
propertyValue = ie.Height;
```
9 Add the following code at the bottom of the function in the **Set code** pane:

```
ie = get(obj, 'Interface');
set(ie, 'Height', propertyValue);
```
10 Save your driver.

Defining Functions

A common function for Internet Explorer is to download a Web page. In the following steps, you create a function called `goTo` that allows you to navigate the Web with the browser.

1 Select the `Functions` node in the driver editor tree.
2 In the **Add function** field, enter the text `goTo`, and click **Add**.
3 Expand the `Functions` node in the tree, and select the new node `goTo`.

Writing functions for generic drivers in the MATLAB Instrument Driver Editor is a matter of writing straight code.

Your `goTo` function requires only one input argument: the URL of the Web page to navigate to. You can call that argument `site`.

4 Change the first line of the MATLAB code pane to read

```
function goTo(obj, site)
```

The variable `obj` is the device object using this driver. The value of `site` is a 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 get a handle to the COM object, then call the IE COM method `Navigate2`, passing to it the value of `site`.

5 Add the following code at the bottom of the function in the MATLAB code pane:

```
ie = obj.Interface;
invoke(ie, 'Navigate2', site);
```
6 Save your driver, and close the MATLAB Instrument Driver Editor.

Now that your generic driver is ready, you can use it with the Test & Measurement Tool (`tmtool`) or at the MATLAB command line.

Using Generic Driver with Test & Measurement Tool

In this section...

Creating and Connecting the Device Object

With the Test & Measurement Tool you can scan for your driver, create a device object that uses that driver, set and get properties of the object, and execute functions.

This example illustrates how to use the generic driver you created in "Writing a Generic Driver" on page 15-3.

1   If your driver is not in the *matlabroot*\toolbox\instrument\instrument \drivers directory, in the MATLAB Command Window, make sure that the directory containing your driver is on the MATLAB path.

    ```
    path
    ```

    If you do not see the directory in the path listing, and the driver is not in the *matlabroot*\toolbox\instrument\instrument\drivers directory, add the directory to the path with the command

    ```
    addpath directory
    ```

    where *directory* is the pathname to the directory containing your driver.

2   Open the Test & Measurement Tool.

    ```
    tmtool
    ```

3   In the Test & Measurement Tool tree, expand the Instrument Drivers node.
4   Select the MATLAB Instrument Drivers node.
5   Your driver might not be listed yet, so click **Scan** in the lower-right corner of the tool. If the tool found your driver, it is listed in the tree as ie_drv.mdd.
6   Select the ie_drv.mdd node in the tree.
7   Right-click the ie_drv.mdd node in the tree, and select **Create Device Object Using Driver**. The following dialog box appears.
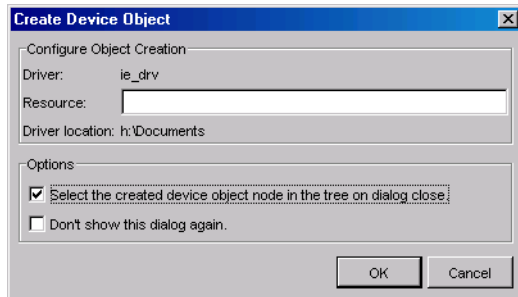
8   Select the **Select the created device object in the tree on dialog close** check box. The device object in this example does not need a resource, so keep that field empty.

9   Click **OK**.

When the Test & Measurement Tool creates the device object, an entry for the object appears as a node in the tree. The `Browser-ie_drv` node should already be selected in the tree. This refers to the device object you just created.

10 Click **Connect** in the upper-right corner of the Test & Measurement Tool. This establishes a communication channel between the tool and the IE browser window, and an empty IE window appears on your screen. Remember that the **Create** code for your driver creates an object for the IE browser, and the **Connect** code and makes its window visible.

Accessing Properties

The driver you created allows you to specify where the browser window appears on your screen and how large it is.

1   Click the **Properties** tab, and then select `Top` in the **Device object properties** list.

The first value displayed for setting this property is `0.0`.

2   Click **Set**. The IE browser window shifts upward to the top edge of your screen.

3   With the mouse, grab the IE window, and drag it down some distance from the top of the screen.

4   Now return to the Test & Measurement Tool window, and click **Get** for the `Top` property. Notice in the **Response** pane how many pixels down you have moved the window.

Use your driver `Vsize` property to change the size of the browser window.

1 Select `Vsize` in the **Device object properties** list.
2 Enter a property value of `200`, and click **Set**. Notice the IE window size.
3 Enter a property value of `400` and click **Set**. Notice the IE window size.
4 Try resizing the IE browser window directly with the mouse. Then in the Test & Measurement Tool, click **Get** for the `Vsize` property. Notice the value returned to the **Response** pane.

Using Functions

Use the `goTo` function of your generic driver to control the Web page that the browser displays.

1 In the Test & Measurement Tool, click the **Functions** tab for your device object.
2 Select `goTo` in the list of **Device object functions**.
3 In the **Input argument(s)** field, enter `'www.mathworks.com'`. Be sure to include the single quotes.
4 Click **Execute**. Observe the IE browser and see that it displays the MathWorks Web site.
5 Experiment freely. When you are finished, right-click the `Browser-ie_drv node` in the tree and select **Delete Object**. Close the Test & Measurement Tool, and close the IE browser window you created in this example.

Using a Generic Driver at Command Line

Creating and Connecting the Device Object

The Instrument Control Toolbox software provides MATLAB commands you can use in the Command Window or in files to create a device object that uses a driver, set and get properties of the object, and execute functions.

This example illustrates how to use the generic driver you created in "Writing a Generic Driver" on page 15-3.

1   If your driver is not in the *matlabroot*\toolbox\instrument\instrument\drivers directory, in the MATLAB Command Window, make sure that the directory containing your driver is on the MATLAB software path.

    path

    If you do not see the directory in the path listing, and the driver is not in the *matlabroot*\toolbox\instrument\instrument\drivers directory, add the directory to the path with the command

    addpath *directory*

    where *directory* is the pathname to the directory containing your driver.

2   Create a device object using your driver. For the driver used in this example, the icdevice function does not require an argument for a resource when using a generic driver. What the object connects to and how it makes that connection are defined in the **Create** code of your driver.

    ie_obj = icdevice('ie_drv');

3   Connect the object.

    connect(ie_obj);

When the device object is connected, an empty IE window appears on your screen. Now you can communicate directly with the IE browser from the MATLAB Command window.

Accessing Properties

The driver you created allows you to specify where the browser window appears on your screen and how large it is. You read and write the properties of your device object with the `get` and `set` functions, respectively.

1 View all of the properties of your device object.

```
get(ie_obj)
    ConfirmationFcn =
    DriverName = ie_drv.mdd
    DriverType = MATLAB generic
    InstrumentModel =
    Interface = [1x1 COM.internetexplorer_application]
    LogicalName =
    Name = Browser-ie_drv
    ObjectVisibility = on
    RsrcName =
    Status = open
    Tag =
    Timeout = 10
    Type = Browser
    UserData = []

    BROWSER specific properties:
    Top = 47
    Vsize = 593
```

2 Most of the properties listed belong to all device objects. For this example, the properties of interest are those listed as `BROWSER specific properties`, that is, `Top` and `Vsize`.

The `Top` property defines the IE browser window position in pixels from the top of the screen. `Vsize` defines the vertical size of the window in pixels.

3 Shift the IE browser window to the top of the screen.

```
ie_obj.Top = 0;
```

4 With the mouse, grab and drag the IE browser window down away from the top of the screen.

5 Find the window's new position by examining the `Top` property.

```
ie_obj.Top
ans =
    120
```

Adjust the size of the window by setting the `Vsize` property.

```
ie_obj.Vsize = 200);
```

6 Make the window larger by increasing the property value.

```
ie_obj.Vsize = 600);
```

Using Functions

By using the `goTo` function of your generic driver, you can control the Web page displayed in the IE browser window.

1 View all of the functions (methods) of your device object.

```
methods(ie_obj)

Methods for class icdevice:

Contents     disp       icdevice      instrnotify methods   size
class        display    igetfield     instrument  ne        subsasgn
close        end        inspect       invoke      obj2mfile subsref
connect      eq         instrcallback isa         open      vertcat
ctranspose   fieldnames instrfind     isequal     openvar
delete       get        instrfindall  isetfield   propinfo
devicereset  geterror   instrhelp     isvalid     selftest
disconnect   horzcat    instrhwinfo   length      set

Driver specific methods for class icdevice:

goTo
```

Most of the methods listed apply to all device objects. For this example, the method of interest is the one listed under `Driver specific methods`, that is, `goTo`.

2 Use the `goTo` function to specify the page for the IE browser to display.

```
invoke(ie_obj, 'goTo', 'www.mathworks.com');
```

If you have access to the Internet, the IE window should display the MathWorks Web site.

3   When you are finished with your example, clean up the MATLAB workspace by removing the object.

```
disconnect(ie_obj);
delete(ie_obj);
clear ie_obj;
```

4   Close the IE browser window you created in this example.

16

Saving and Loading the Session

This chapter describes how to save and load information associated with an instrument control session.

- "Saving and Loading Instrument Objects" on page 16-2
- "Debugging: Recording Information to Disk" on page 16-6

Saving and Loading Instrument Objects

In this section...
"Saving Instrument Objects to a File" on page 16-2
"Saving Objects to a MAT-File" on page 16-3

Saving Instrument Objects to a File

You can save an instrument object to a file using the `obj2mfile` function. `obj2mfile` provides you with these options:

-   Save all property values or save only those property values that differ from their default values.

    Read-only property values are not saved. Therefore, read-only properties use their default values when you load the instrument object into the MATLAB workspace. To determine if a property is read-only, use the `propinfo` function or examine the property reference pages.

-   Save property values using the `set` syntax or the dot notation.

If the `UserData` property is not empty, or if a callback property is set to a cell array of values or a function handle, then the data stored in these properties is written to a MAT-file when the instrument object is saved. The MAT-file has the same name as the file containing the instrument object code.

For example, suppose you create the GPIB object `g`, return instrument identification information to the variable `out`, and store `out` in the `UserData` property.

```
g = gpib('ni',0,1);
g.Tag = 'My GPIB object';
fopen(g)
cmd = '*IDN?';
fprintf(g,cmd)
out = fscanf(g);
g.UserData = out;
```

The following command saves `g` and the modified property values to the file `mygpib.m`. Because the `UserData` property is not empty, its value is automatically written to the MAT-file `mygpib.mat`.

```
obj2mfile(g,'mygpib.m');
```

Use the `type` command to display `mygpib.m` at the command line.

Loading the Instrument Object

To load an instrument object that was saved as a file into the MATLAB workspace, type the name of the file at the command line. For example, to load `g` from the file `mygpib.m`,

```
g = mygpib
```

The display summary for `g` is shown below. Note that the read-only properties such as `Status`, `BytesAvailable`, `ValuesReceived`, and `ValuesSent` are restored to their default values.

```
GPIB Object Using NI Adaptor : GPIB0-1

Communication Address
   BoardIndex:        0
   PrimaryAddress:    1
   SecondaryAddress:  0

Communication State
   Status:            closed
   RecordStatus:      off

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

When loading `g` into the workspace, the MAT-file `mygpib.mat` is automatically loaded and the `UserData` property value is restored.

```
g.UserData
ans =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

Saving Objects to a MAT-File

You can save an instrument object to a MAT-file just as you would any workspace variable — using the `save` command. For example, to save the GPIB object `g` and

the variables cmd and out, defined in "Saving Instrument Objects to a File" on page 16-2, 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:        O
   PrimaryAddress:    1
   SecondaryAddress:  O

Communication State
   Status:            closed
   RecordStatus:      off
```

```
Read/Write State
   TransferStatus:    idle
   BytesAvailable:    O
   ValuesReceived:    O
   ValuesSent:        O
```

Debugging: Recording Information to Disk

In this section...
"Using the record Function" on page 16-6
"Introduction to Recording Information" on page 16-7
"Creating Multiple Record Files" on page 16-7
"Specifying a File Name" on page 16-7
"Record File Format" on page 16-8
"Recording Information to Disk" on page 16-10

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

Property Name	Description
RecordDetail	Specify the amount of information saved to a record file.
RecordMode	Specify whether data and event information are saved to one record file or to multiple record files.
RecordName	Specify the name of the record file.
RecordStatus	Indicate if data and event information are saved to a record file.

Introduction to Recording Information

This example creates the GPIB object g, records the number of values transferred between g and the instrument, and stores the information to the file text myfile.txt.

```
g = gpib('ni',0,1);
g.RecordName = 'myfile.txt';
fopen(g)
record(g)
fprintf(g,'*IDN?')
out = fscanf(g);
```

End the instrument control session.

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

Use the type command to display myfile.txt at the command line.

Creating Multiple Record Files

When you initiate recording with the record function, the RecordMode property determines if a new record file is created or if new information is appended to an existing record file.

You can configure RecordMode to overwrite, append, or index. If RecordMode is overwrite, then the record file is overwritten each time recording is initiated. If RecordMode is append, then the new information is appended to the file specified by RecordName. If RecordMode is index, a different disk file is created each time recording is initiated. The rules for specifying a record file name are discussed in "Specifying a File Name" on page 16-7.

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

`value = (-1)`$^{\text{sign}}$`(2`$^{\text{exp-127}}$`)(1.significand)`

Double-precision floating-point values consist of 64 bits, and the value is given by

`value = (-1)`$^{\text{sign}}$`(2`$^{\text{exp-1023}}$`)(1.significand)`

The floating-point format component and the associated single-precision and double-precision bits are given below.

Format Component	Single-Precision Bits	Double-Precision Bits
`sign`	1	1

Format Component	Single-Precision Bits	Double-Precision Bits
`exp`	2-9	2-12
`significand`	10-32	13-64

For example, suppose you record the decimal value 4.25 using the single-precision format. The record file stores 4.25 as the hex value 40880000, which is calculated from the IEEE single-precision floating-point format. To reconstruct the original value, convert the hex value to a decimal value using `hex2dec`:

```
dval = hex2dec('40880000')
dval =
    1.082654720000000e+009
```

Convert the decimal value to a binary value using `dec2bin`:

```
bval = dec2bin(dval,32)
bval =
01000000100010000000000000000000
```

The interpretation of `bval` is given by the preceding table. The left most bit indicates the value is positive because $(-1)^0 = 1$. The next 8 bits correspond to the exponent, which is given by

```
exp = bval(2:9)
exp =
10000001
```

The decimal value of `exp` is $2^7 + 2^0 = 129$. The remaining bits correspond to the significant, which is given by

```
significand = bval(10:32)
significand =
00010000000000000000000
```

The decimal value of `significand` is $2^{-4} = 0.0625$. You reconstruct the original value by plugging the decimal values of `exp` and `significand` into the formula for IEEE singles:

```
value = (-1)^0(2^129 - 127)(1.0625)
value = 4.25
```

Recording Information to Disk

This example extends "Reading and Writing 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.

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

2 **Configure properties** — Configure the input buffer to accept a reasonably large number of bytes, and configure the timeout value to two minutes to account for slow data transfer.

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

Configure g to execute the callback function instrcallback every time 5000 bytes are stored in the input buffer.

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

Configure g to record information to multiple disk files using the verbose format. The first disk file is defined as WaveForm1.txt.

```
g.RecordMode = 'index';
g.RecordDetail = 'verbose';
g.RecordName = 'WaveForm1.txt';
```

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

```
fopen(g)
```

4 **Write and read data** — Initiate recording.

```
record(g)
```

Configure the scope to transfer the screen display as a bitmap.

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

Initiate the asynchronous read operation, and begin generating events.

```
readasync(g)
```

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

```
BytesAvailable event occurred at 09:04:33 for the object: GPIB0-1.
BytesAvailable event occurred at 09:04:42 for the object: GPIB0-1.
BytesAvailable event occurred at 09:04:51 for the object: GPIB0-1.
BytesAvailable event occurred at 09:05:00 for the object: GPIB0-1.
BytesAvailable event occurred at 09:05:10 for the object: GPIB0-1.
BytesAvailable event occurred at 09:05:19 for the object: GPIB0-1.
BytesAvailable event occurred at 09:05:28 for the object: GPIB0-1.
```

Wait until all the data is stored in the input buffer, and then transfer the data to the MATLAB workspace as unsigned 8-bit integers.

```
out = fread(g,g.BytesAvailable,'uint8');
```

Toggle the recording state from `on` to `off`. Because the `RecordMode` value is `index`, the record file name is automatically updated.

```
record(g)
g.RecordStatus
ans =
off
g.RecordName
ans =
WaveForm2.txt
```

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

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

The Record File Contents

To display the contents of the WaveForm1.txt record file,

```
type WaveForm1.txt
```

The record file contents are shown below. Note that data returned by the `fread` function is in hex format (most of the bitmap data is not shown).

```
Legend:
  * - An event occurred.
  > - A write operation occurred.
  < - A read operation occurred.

1     Recording on 18-Jun-2000 at 09:03:53.529. Binary data in
      little endian format.
2   > 18 ascii values.
      HARDCOPY:PORT GPIB
3   > 19 ascii values.
      HARDCOPY:FORMAT BMP
4   > 14 ascii values.
      HARDCOPY START
5   * BytesAvailable event occurred at 18-Jun-2000 at 09:04:33.334
6   * BytesAvailable event occurred at 18-Jun-2000 at 09:04:41.775
7   * BytesAvailable event occurred at 18-Jun-2000 at 09:04:50.805
8   * BytesAvailable event occurred at 18-Jun-2000 at 09:04:00.266
9   * BytesAvailable event occurred at 18-Jun-2000 at 09:05:10.306
10  * BytesAvailable event occurred at 18-Jun-2000 at 09:05:18.777
11  * BytesAvailable event occurred at 18-Jun-2000 at 09:05:27.778
12  < 38462 uint8 values.
      42 4d cf 03 00 00 00 00 00 00 3e 00 00 00 28 00
      00 00 80 02 00 00 e0 01 00 00 01 00 01 00 00 00
      00 00 00 96 00 00 00 00 00 00 00 00 00 00 00 00
      .
      .
      .
      ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
      ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
      ff ff ff ff ff ff ff ff ff ff ff ff ff ff
13  Recording off.
```

Test & Measurement Tool

This chapter describes how to use the Test & Measurement Tool to access your hardware interfaces and instrument drivers.

- "Test & Measurement Tool Overview" on page 17-2
- "Using the Test & Measurement Tool" on page 17-4

Test & Measurement Tool Overview

In this section...
"Instrument Control Toolbox Software Support" on page 17-2
"Navigating the Tree" on page 17-2

Instrument Control Toolbox Software Support

The Test & Measurement Tool (tmtool) enables you to configure and control resources (instruments, serial devices, drivers, interfaces, etc.) accessible through the toolbox without having to write the MATLAB script.

You can use the Test & Measurement Tool to manage your session with the toolbox. This tool enables you to do the following:

- Detect available hardware and drivers.
- Connect to an instrument or device.
- Configure instrument or device settings.
- Read and write data.
- Automatically generate the MATLAB script.
- Visualize acquired data.
- Export acquired data to the MATLAB workspace.

Navigating the Tree

You start the Test & Measurement Tool by typing

```
tmtool
```

You navigate to the various hardware control panes using the tool's tree. Start by selecting the toolbox you want to work with, which displays a set of instructions in the right pane. These instructions explain the basic steps to establishing communication with an instrument.

For example, the following figure shows the pane displayed when you select Instrument Control Toolbox.
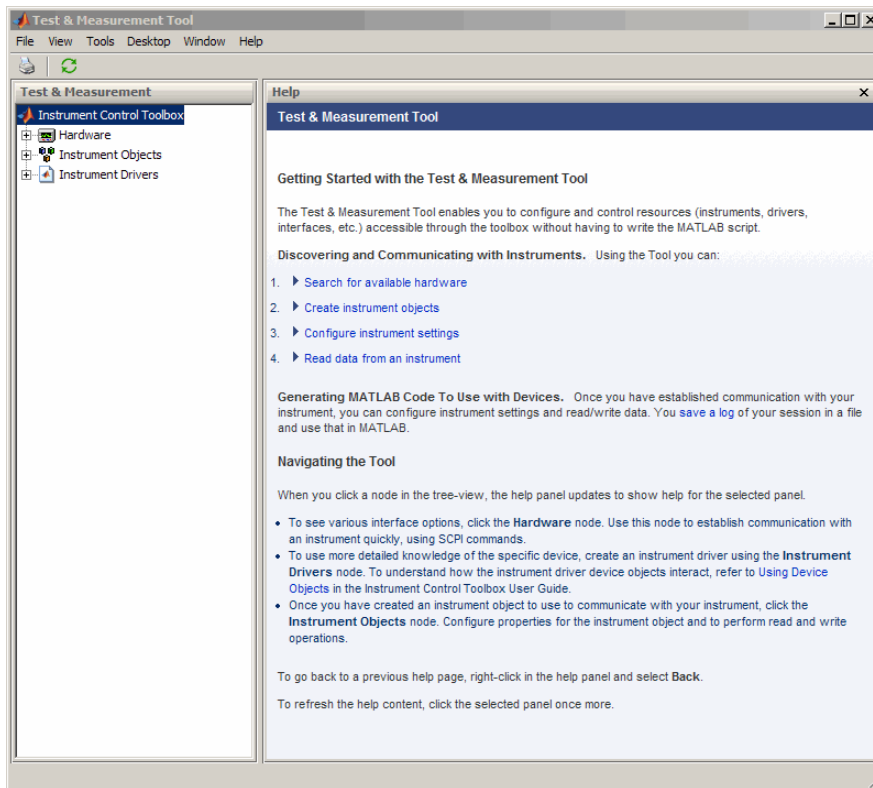
Using the Test & Measurement Tool

In this section...
"Overview of the Examples" on page 17-4
"Hardware" on page 17-4
"Instrument Objects" on page 17-11
"Instrument Drivers" on page 17-16

Overview of the Examples

These examples illustrates a typical session using the Test & Measurement Tool for instrument control. The session entails communicating with a Tektronix TDS 210 oscilloscope via a GPIB interface.

To start the tool, on the MATLAB Command window, type:

```
tmtool
```

Hardware

When the tool displays, expand (click the **+**) the `Instrument Control Toolbox` node in the tree. Next, expand the `Hardware` node. The tree now looks like this.
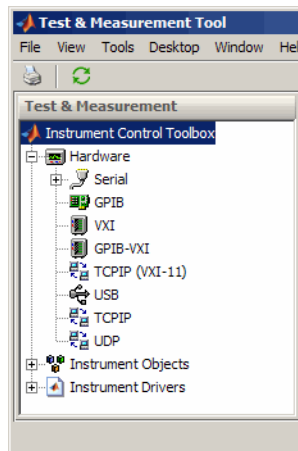
Selecting the Interface and Scanning for GPIB Boards

Next, scan for installed GPIB boards by selecting the `GPIB` node. The right pane changes to the **Installed GPIB Board** list. Click **Scan** to see what boards are installed. The following figure shows the scan result from a system with one Capital Equipment Corp and one Keithley GPIB board.
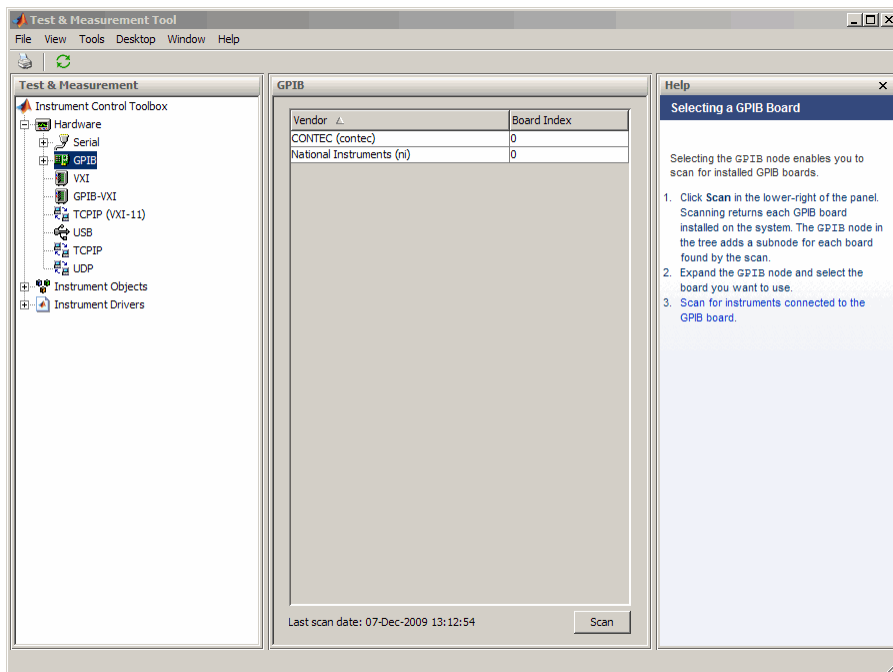
Scanning for Instruments Connected to GPIB Boards

After determining what GPIB boards are installed, you must determine what instruments are connected to those boards. Expand the `GPIB` node and select a board.

The right pane changes to the **GPIB Instruments** list. Click **Scan** to see what instruments are connected to this board. The following figure shows the scan result from a system with a Tektronix TDS 210 connected at primary address 8`.
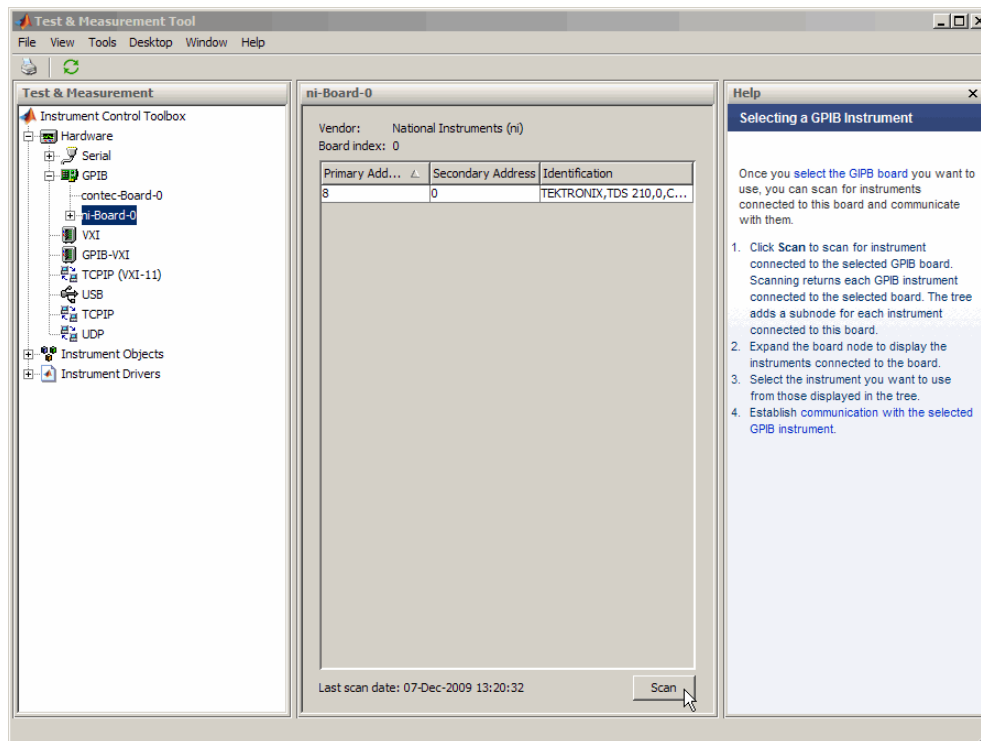
Configuring the Interface

You can change the configuration of the interface by clicking the **Configure** tab. This pane displays properties you can set to configure the instrument communication settings. In the following view of the **Configure** pane, the `Timeout` property value has been set to `10` seconds.
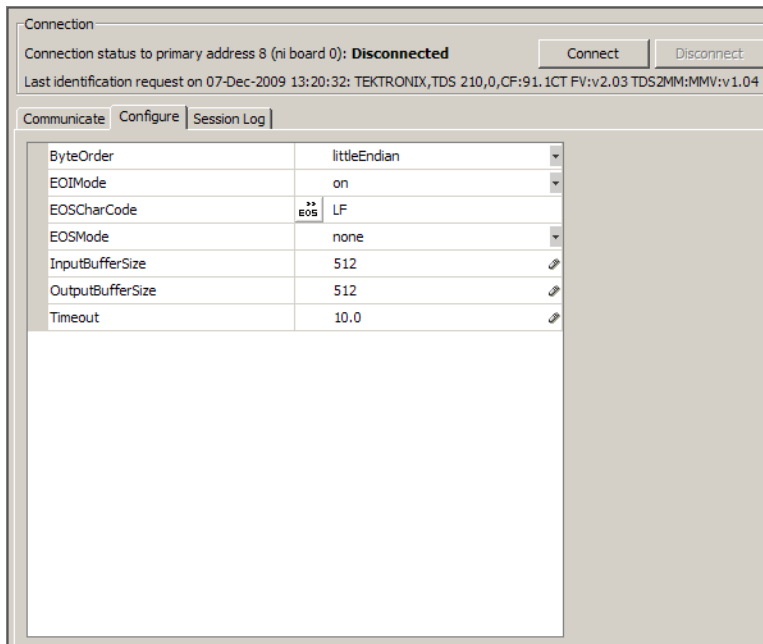
Establishing the Connection

Expand the `ni-Board-0` node and select the instrument at primary address 4: `PAD-8` `(TEKTRONIX,TDS 210...)`. The right pane changes to the control pane you use for writing and reading data to and from that instrument.
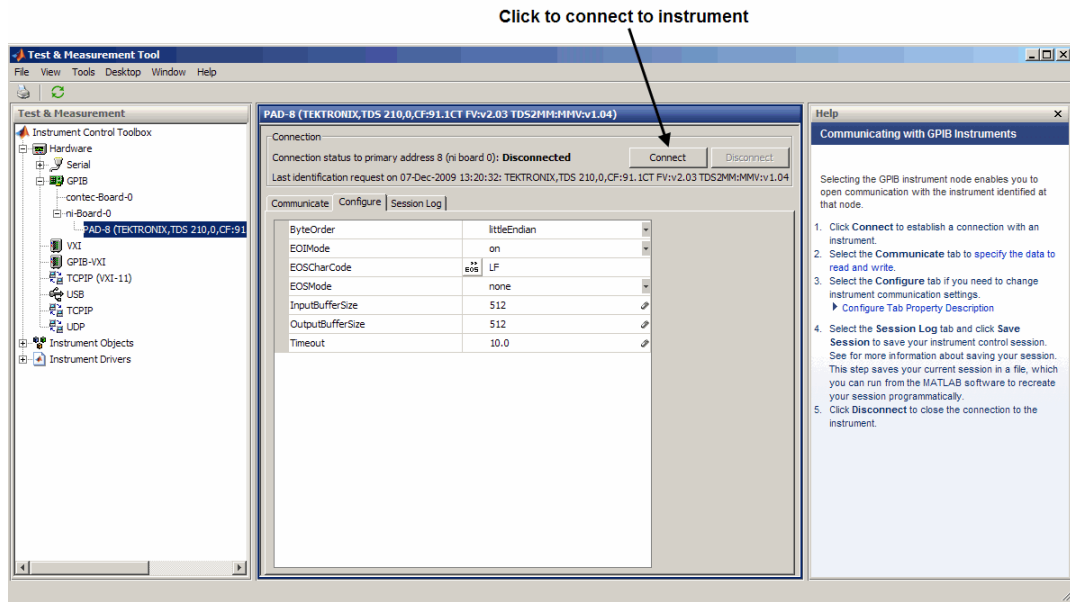
Click **Connect** to establish communication with the instrument. The tool creates an interface object representing the communication channel to the instrument.

Writing and Reading Data

Selecting the **Communicate** tab displays the pane you use to write and read data. You can write and read data separately using the **Write** and **Read** buttons, or you can use the **Query** button to write and read in a single operation.

The following figure shows the pane after a brief session involving the following steps:

1   Open communication with the instrument.

2   Enter `*IDN?` as **Data to Write**, and click **Query** (write/read). This executes the identify command.

3   Enter `CURVE?` as **Data to Write**, and click **Query**. This retrieves the waveform data from the scope.
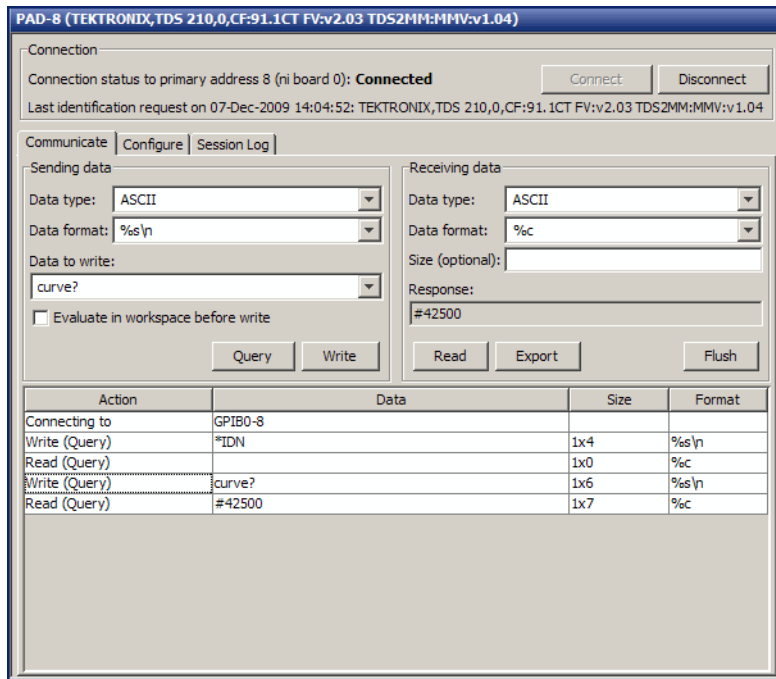
Exporting Instrument Data

You can export the data acquired from instruments to any of the following:

- MATLAB workspace as a variable
- Figure window as a plot
- MAT-file for storage in a file
- The MATLAB Variables editor for modification

To export data, select **File** > **Export** > **Instrument Response(s)** from the menu bar. When the Data Exporter dialog box opens, choose the variables to export. The following figure shows the Data Exporter set to export the curve data to the MATLAB workspace as the variable `data2`.
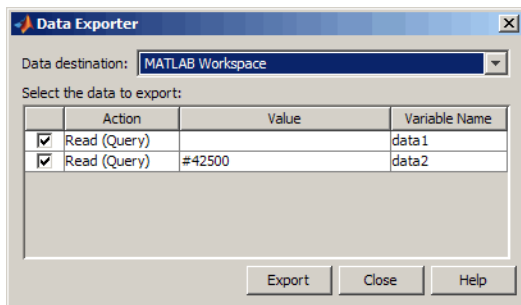
Note: If you repeatedly generate a large amount of data in the Test and Measurement tool, you must delete the data object after you export it to MATLAB. This will allow the tool to return resources to MATLAB correctly and will prevent MATLAB from failing to respond the next time you acquire data.

Exporting the GPIB Object

When you open a connection to an instrument, the Test & Measurement Tool creates an instrument object automatically. You can export the GPIB instrument object created in this example as any of the following:

- MATLAB workspace object that you can use as an argument in instrument control commands

- File containing the call to the GPIB constructor and the commands to set object properties

- MAT-file for storage in a file

To export the object, select **File** > **Export** > **Instrument Object** from the menu bar. When the Object Exporter dialog box opens, choose the object to export. The following figure shows the Object Exporter set to export the object to a file. (When you run that file, it creates a new object with the equivalent settings.)
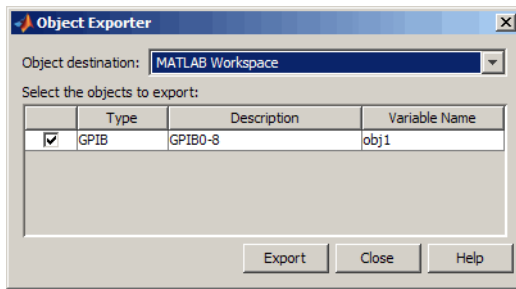
Saving Your Instrument Control Session

The **Session Log** tab displays the code equivalent of your instrument control session. You can save this code to a file so that you can execute the same commands programmatically.

Select **File > Save Session Log** from the menu bar or click **Save Session**. From this dialog box you can specify a file name and directory location for the file.

Instrument Objects

Interface Objects

The Test & Measurement Tool creates interface objects automatically when you open a communication channel to an instrument by clicking the **Communication Status** button. To explicitly create and configure an interface object:

1   Expand the `Instrument Objects` node in the tree, and select `Interface Objects`. The **Interface Objects** pane appears on the right.
2   Click **New Object** to open the New Object Creation dialog box.
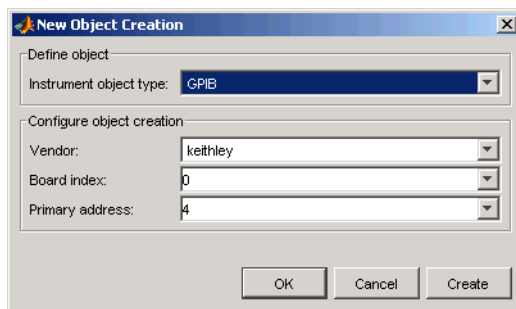
3 Specify the object parameters and click **OK** to create the new object.

Device Objects

To create and configure a device object:

1 Expand the `Instrument Objects` node in the tree, and select `Device Objects`. The **Device Objects** pane appears on the right.

2 Click **New Object** to open the New Object Creation dialog box. In this case, the **Instrument object type** is already set for `device`.
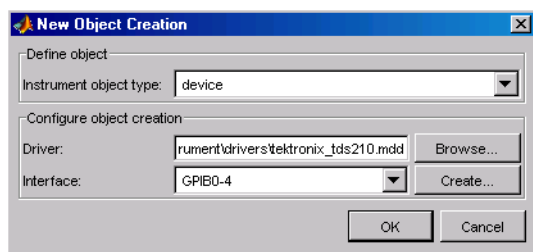
3 Specify or browse for the instrument driver you want to use; then choose from among the available interface objects, or create one if necessary.

4 Click **OK** to create the new device object.

Setting Instrument Object Properties

Whether the instrument objects are created automatically, created through the New Object Creation dialog box, or created on the MATLAB Command window, the Test & Measurement Tool enables you to set the properties of these objects. To change object properties in the Test & Measurement Tool:

1 Expand the `Instrument Objects` node in the tree, then either `Interface Objects` or `Device Objects`, and select the object whose properties you want to set.

2 Click the **Configure** tab in the right pane.

3 Set properties displayed in this pane, as shown in the following figures.
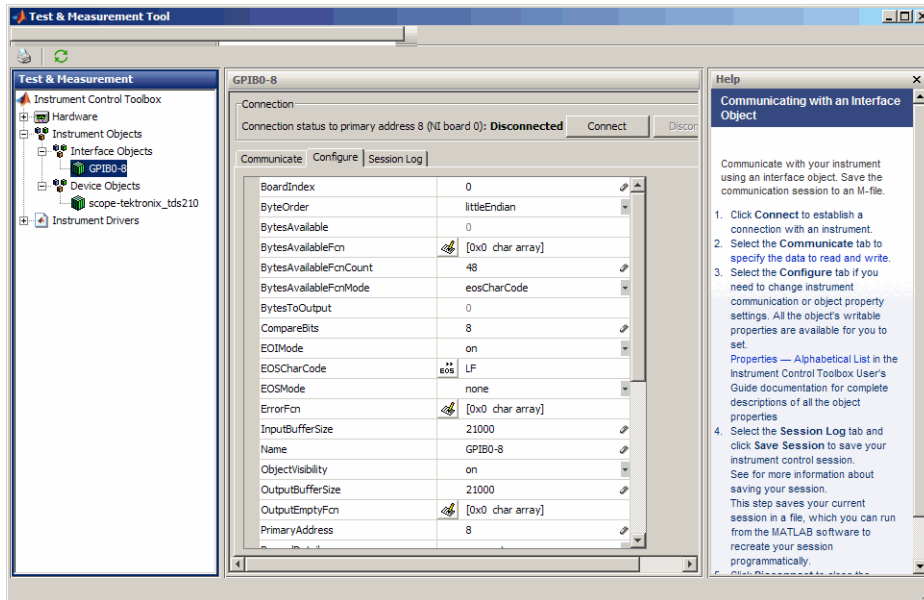
Configuring Interface Object Properties
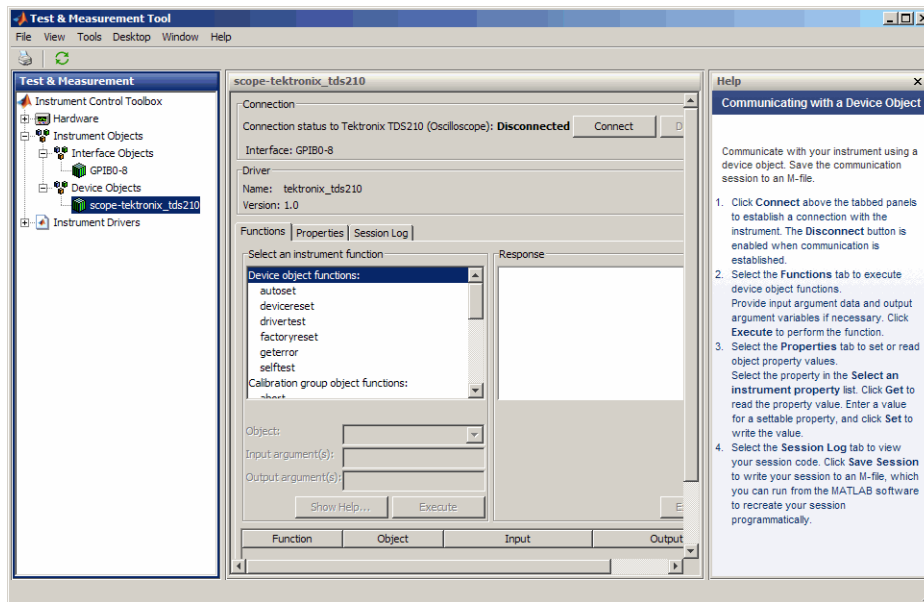
Configuring Device Object Properties

Communicating with Your Instrument

Using an Interface Object

When communicating with your instrument using an interface object, you send data to instrument in the form of raw instrument commands. In the following figure, the Test & Measurement Tool sends the `*RST` string to the TDS 210 oscilloscope via an interface object. `*RST` is the oscilloscope's reset command.
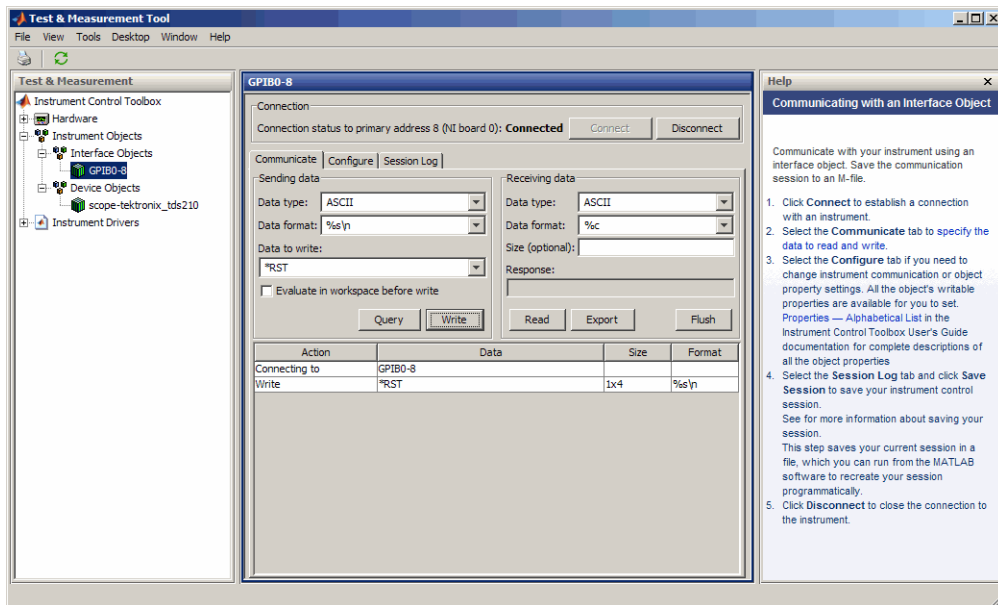
Communicating via an Interface Object

Using a Device Object

When communicating with your instrument using a device object, instead of employing instrument commands, you invoke device object methods (functions) or you set device object properties as provided by the MATLAB instrument driver for that instrument.

In the following figure, the Test & Measurement Tool resets a TDS 210 oscilloscope by issuing a call to the `devicereset` function of the instrument driver. Communicating this way, you don't need to know what the actual oscilloscope reset command is.
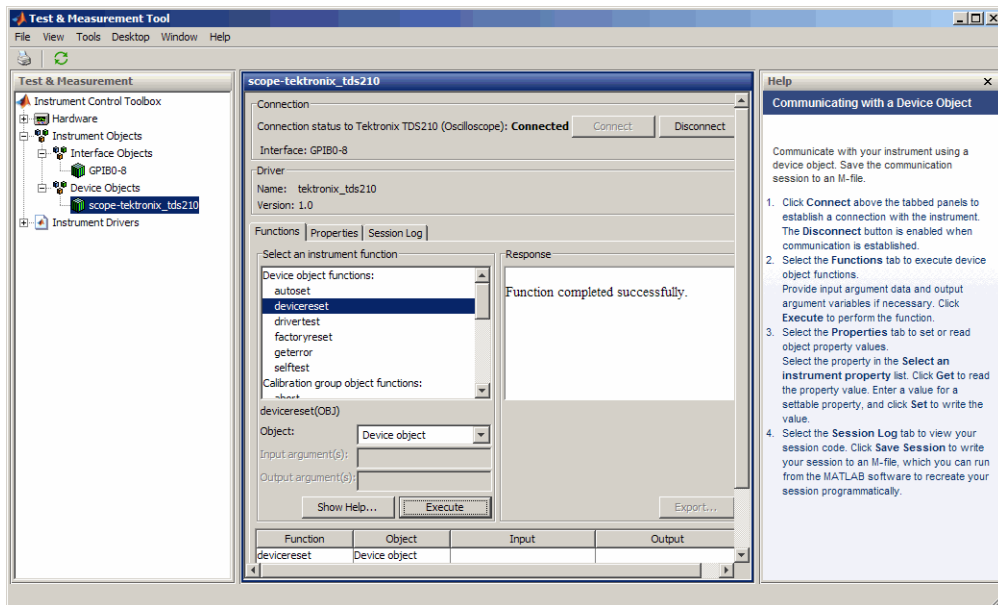
Communicating via a Device Object

Instrument Drivers

The Test & Measurement Tool enables you to scan for installed drivers, and to use those drivers when creating device objects.

MATLAB Instrument Drivers

MATLAB instrument drivers include

- MATLAB interface drivers
- MATLAB VXI*plug&play* drivers
- MATLAB IVI drivers

Select the `MATLAB Instrument Drivers` node in the tree. Then click **Scan** to get an updated display of all the installed MATLAB instrument drivers found on the MATLAB software path.
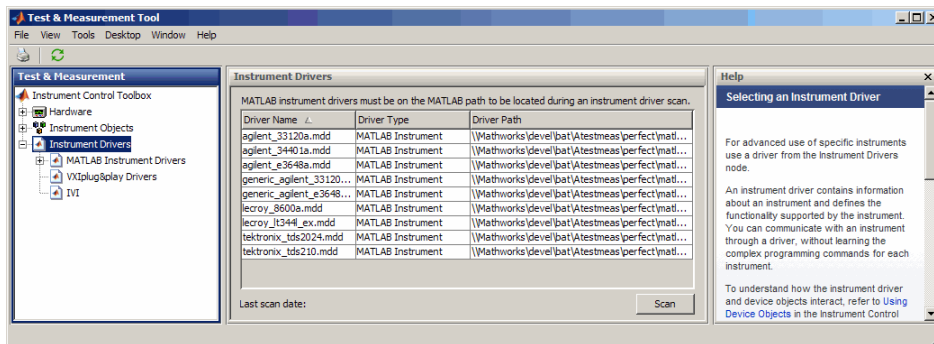
When the Test & Measurement Tool scans for drivers, it makes them available as nodes under the driver type node. Expand the MATLAB software Instrument Drivers node to reveal the installed drivers. Select one of them to see the driver's details.
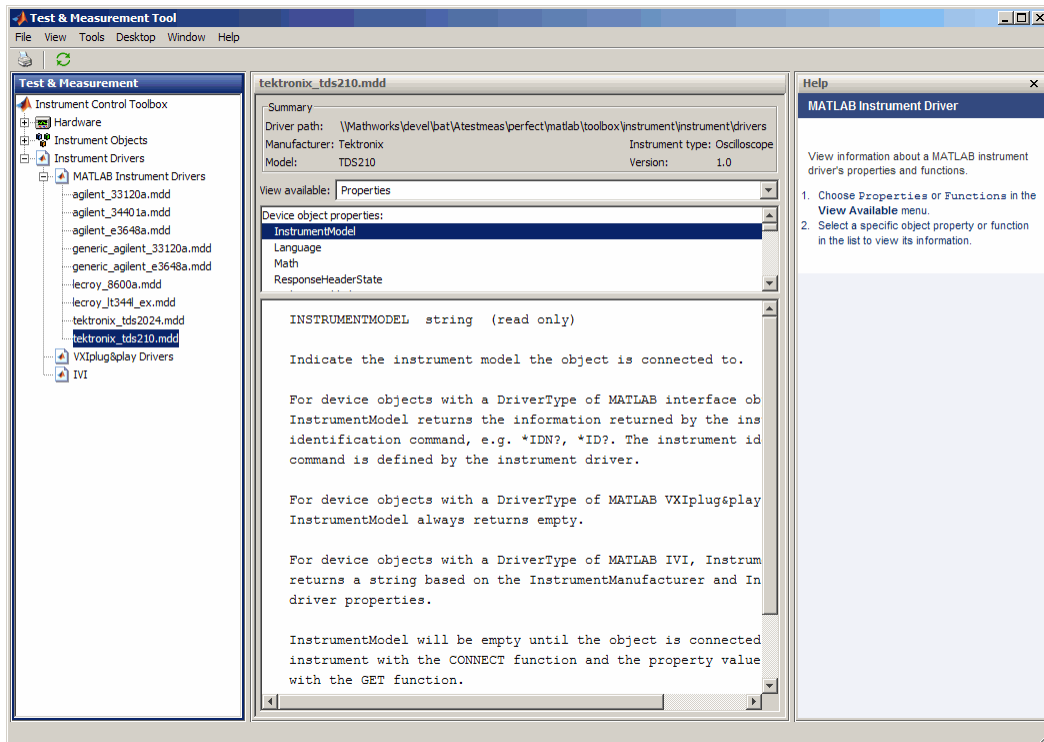
You can choose to see the driver's properties or functions. When you select the particular property or function, the tool displays that item's description.

VXI*plug&play* Drivers

For an example of scanning for installed VXI*plug&play* drivers with the Test & Measurement Tool, see "VXI plug and play Drivers" on page 12-4.

IVI Drivers

For an example of scanning for installed IVI-C or IVI-COM drivers with the Test & Measurement Tool, see "Getting Started with IVI Drivers" on page 13-6. For using the Test & Measurement Tool to examine or configure an IVI configuration store, see "Configuring an IVI Configuration Store" on page 13-17.

Using the Instrument Driver Editor

This chapter describes how to use the Instrument Driver Editor to create, import, or modify instrument drivers.

MATLAB Instrument Driver Editor Overview

In this section...
"What Is a MATLAB Instrument Driver?" on page 18-2
"How Does a MATLAB Instrument Driver Work?" on page 18-3
"Why Use a MATLAB Instrument Driver?" on page 18-3

What Is a MATLAB Instrument Driver?

The Instrument Control Toolbox software provides the means of communicating directly with a hardware instrument through an interface object. If you are programming directly through an interface object, you need to program with the command language of the instrument itself. Any substitution of instrument, such as make or model, may require a change to the appropriate the MATLAB code.
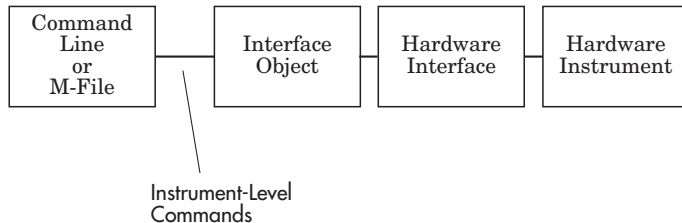
A MATLAB instrument driver offers a layer of interpretation between you and the instrument. The instrument driver contains all the necessary commands for programming the instrument, so that you do not need to be aware of the specific instrument commands. Instead, you can program the instrument with familiar or consistent device object properties and functions.

The following figure shows how a device object and instrument driver offer a layer between the command line and the interface object. The instrument driver handles the instrument-level commands, so that as you program from the command line, you need only manipulate device object properties and functions, rather than instrument commands.
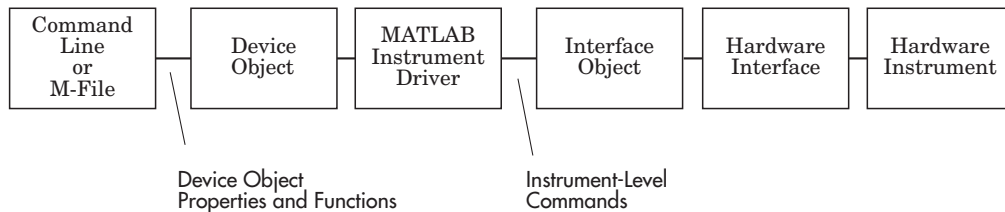
In addition to containing instrument commands, the instrument driver can also contain the MATLAB code to provide analysis based upon instrument setup or data.

Note: For many instruments, a MATLAB instrument driver already exists and you will not need to create a MATLAB instrument driver for your instrument. For other instruments, there may be a similar MATLAB instrument driver and you will need to edit it. If you would like more information on how to edit a MATLAB instrument driver, you may want to begin with "Modifying MATLAB Instrument Drivers" on page 18-59.

Note: The Instrument Driver Editor is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

How Does a MATLAB Instrument Driver Work?

A MATLAB instrument driver contains information on the functionality supported by an instrument. You access this functionality through a device object's properties and functions.

When you query or configure a property of the device object using the `get` or `set` function, or when you call (`invoke`) a function on the device object, the MATLAB instrument driver provides a translation to determine what instrument commands are written to the instrument or what the MATLAB code is executed.

Why Use a MATLAB Instrument Driver?

Using a MATLAB instrument driver isolates you from the instrument commands. Therefore, you do not need to be aware of the instrument syntax, but can use the same code for a variety of related instruments, ignoring the differences in syntax from one instrument to the next.

For example, suppose you have two different oscilloscopes in your shop, each with its own set of commands. If you want to perform the same tasks with the two different instruments, you can create an instrument driver for each scope so that you can control each with the same code. Then substitution of one instrument for another does not require a change in the MATLAB code being used to control it, but only a substitution of the instrument driver.

Creating MATLAB Instrument Drivers

Driver Components

A MATLAB instrument driver contains information about an instrument and defines the functionality supported by the instrument.

Driver Component	Description
Driver Summary and Common Commands	Basic information about the instrument, e.g., manufacturer or model number.
Initialization and Cleanup	Code that is executed at various stages in the instrument control session, e.g., code that is executed upon connecting to the instrument.
Properties	A property is generally used to configure or query an instrument's state information.
Functions	A function is generally used to control or configure an instrument.
Groups	A group combines common functionality of the instrument into one component.

Depending on the instrument and the application for which the driver is being used, all components of the driver may not be defined. You can define the necessary driver components needed for your application with the MATLAB Instrument Driver Editor.

Note: The Instrument Driver Editor is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

MATLAB Instrument Driver Editor Features

The MATLAB Instrument Driver Editor is a tool that creates or edits a MATLAB instrument driver. Specifically, it allows you to do the following:

· Add/remove/modify properties.

· Add/remove/modify functions .

· Define the MATLAB code to wrap around commands sent to instrument.

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 Common Commands

You can assign basic information about the instrument to the MATLAB instrument driver. Summary information can be used to identify the MATLAB instrument driver and the instrument that it represents. Common commands can be used to reset, test, and read error messages from the instrument. Together, this information can be used to initialize and verify the instrument.

Topics in this section include

· "Driver Summary" on page 18-7

· "Common Commands" on page 18-7

· "Defining Driver Summary and Common Commands" on page 18-7

· "Verifying Driver Summary and Common Commands" on page 18-8

Driver Summary

You can assign basic information that describes your instrument in the instrument driver. This information includes the manufacturer of the instrument, the model number of the instrument and the type of the instrument. A version can also be assigned to the driver to assist in revision control.

Common Commands

You can define basic common commands supported by the instrument. The common commands can be accessed through device object properties and functions.

Common Commands	Accessed with Device Object's	Example Instrument Command	Description
Identify	`InstrumentModel` property	`*IDN?`	Returns the identification string of the instrument
Reset	`devicereset` function	`*RST`	Returns the instrument to a known state
Self test	`selftest` function	`*TST?`	Tests the instrument's interface
Error	`geterror` function	`ErrLog:Next?`	Retrieves the next instrument error message

The MATLAB Instrument Driver Editor assigns default values for the Common commands. The common commands should be modified appropriately to match the instrument's command set.

Defining Driver Summary and Common Commands

This example defines the basic driver information and Common commands for a Tektronix TDS 210 oscilloscope using the MATLAB Instrument Driver Editor:

1 Select the `Summary` node in the tree.
2 In the **Driver summary** pane:

   a   Enter `Tektronix` in the **Manufacturer** field.
   b   Enter `TDS 210` in the **Model** field.
   c   Select `Oscilloscope` in the **Instrument type** field.
   d   Enter `1.0` in the **Driver version** field.
3 In the **Common commands** pane:

      a   Leave the **Identify** field with `*IDN?`.

      b   Leave the **Reset** field with `*RST`.

      c   Leave the **Self test** field with `*TST?`

      d   Update the **Error** field with `ErrLog:Next?`

4   Click the **Save** button. Specify the name of the instrument driver as `tektronix_tds210_ex.mdd`.
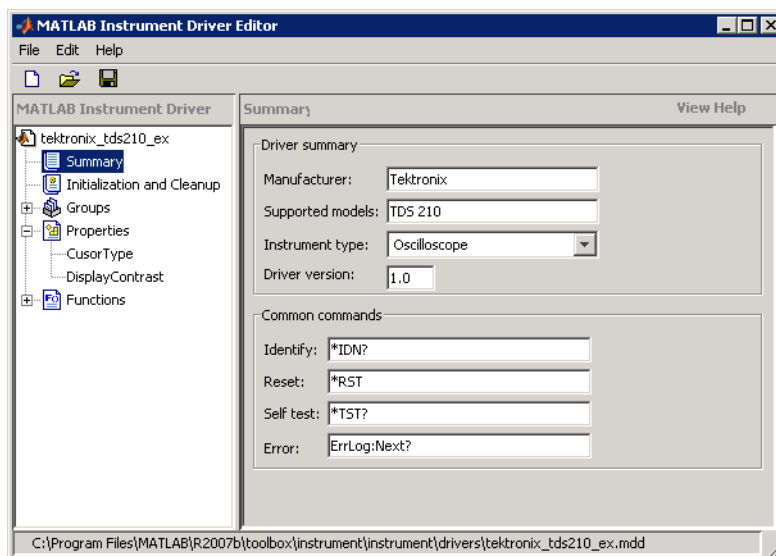
Note For additional information on instrument driver nomenclature, refer to "Saving MATLAB Instrument Drivers" on page 18-6.

Verifying Driver Summary and Common Commands

This procedure verifies the summary information defined in the Driver Summary and Common commands panes. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Keithley GPIB board at board index 0. From the MATLAB Command Window,

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 defined driver information.

```
      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        tekronix_tds210_ex.mdd
          DriverVersion     1.0

      Communication State
          Status            closed

      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 Common commands.

```
      obj.InstrumentModel
      ans =
          TEKTRONIX,TDS 210,0,CF:91.1CT FV:v2.03 TDS2MM:MMV:v1.04

      devicereset(obj)
      selftest(obj)
      ans =
          0

      geterror(obj)
      ans =
          ''
```

5 Disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

Initialization and Cleanup

This section describes how to define code that is executed at different stages in the instrument control session, so that the instrument can be set to a desired state at particular times. Specifically, you can define code that is executed after the device object is created, after the device object is connected to the instrument, or before the device object is disconnected from the instrument. Depending on the stage, the code can be defined as a list of instrument commands that will be written to the instrument or as MATLAB code.

Topics in this section include

- Definitions of the types of code that can be defined
- Examples of code for each supported stage
- Steps used to verify the code

Create Code

You define create code to ensure that the device object is configured to support the necessary properties and functions:

- Create code is evaluated immediately after the device object is created.
- Create code can only be defined as a MATLAB software code.

Defining Create Code

This example defines the create code that ensures that the device object can transfer the maximum waveform size, 2500 data points, supported by the Tektronix TDS 210 oscilloscope. In the MATLAB instrument driver editor,

1 Select the `Initialization and Cleanup` node in the tree.
2 Click the **Create** tab and enter the MATLAB software code to execute on device object creation.

```
% Get the interface object and disconnect from instrument.
```

```
g = obj.Interface;
fclose(g);

% Configure the interface object's buffers to handle up to
% 2500 points (two bytes per point requires 5000 bytes).
g.InputBufferSize = 5000;
g.OutputBufferSize = 5000;
```

3 Click the **Save** button.
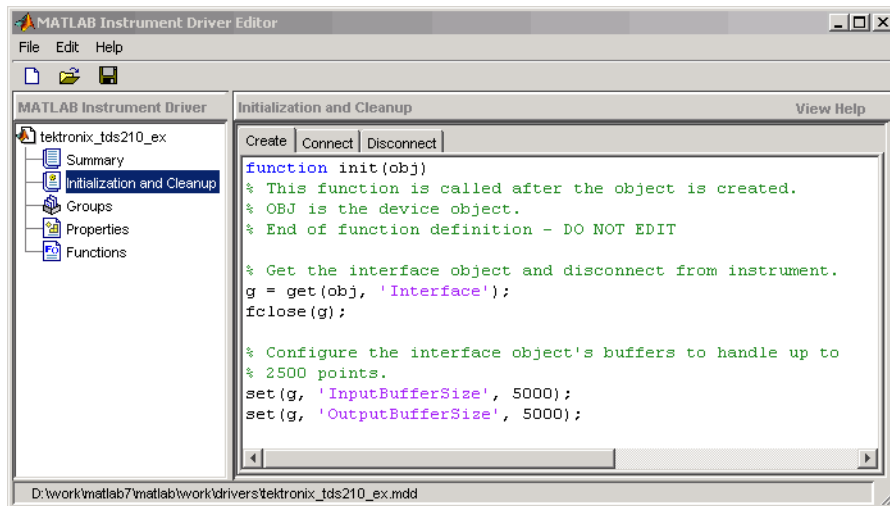
Verifying Create Code

This procedure verifies the MATLAB software create code defined. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a 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 = gpib('keithley', 0, 2);
g.InputBufferSize
ans =
    512

g.OutputBufferSize
ans =
    512
```

18-11

2 Create the device object, `obj`, using the `icdevice` function.

```
obj = icdevice('tektronix_tds210_ex.mdd', g);
```

3 Verify the create code by querying the interface object's buffer sizes.

```
g.InputBufferSize
ans =
    5000

g.OutputBufferSize
ans =
    5000
```

4 Delete the objects.

```
delete([obj g])
```

Connect Code

In most cases you need to know the state or configuration of the instrument when you connect the device object to it. You can define connect code to ensure that the instrument is properly configured to support the device object's properties and functions.

Connect code is evaluated immediately after the device object is connected to the instrument with the `connect` function. The connect code can be defined as a series of instrument commands that will be written to the instrument or as MATLAB software code.

Defining Connect Code

This example defines the connect code that ensures the Tektronix TDS 210 oscilloscope is configured to support the device object properties and functions. Specifically, the instrument will be returned to a known set of instrument settings (instrument reset) and the instrument will be configured to omit headers on query responses.

1 From the MATLAB instrument driver editor, select the `Initialization and Cleanup` node in the tree.

2 Click the **Connect** tab and enter the instrument commands to execute when the device object is connected to the instrument.

- Select `Instrument Commands` from the **Function style** menu.
- Enter the `*RST` command in the **Command** text field, and then click **Add**.
- Enter the `HEADER OFF` command in the **Command** text field, and then click **Add**.

3 Click the **Save** button.
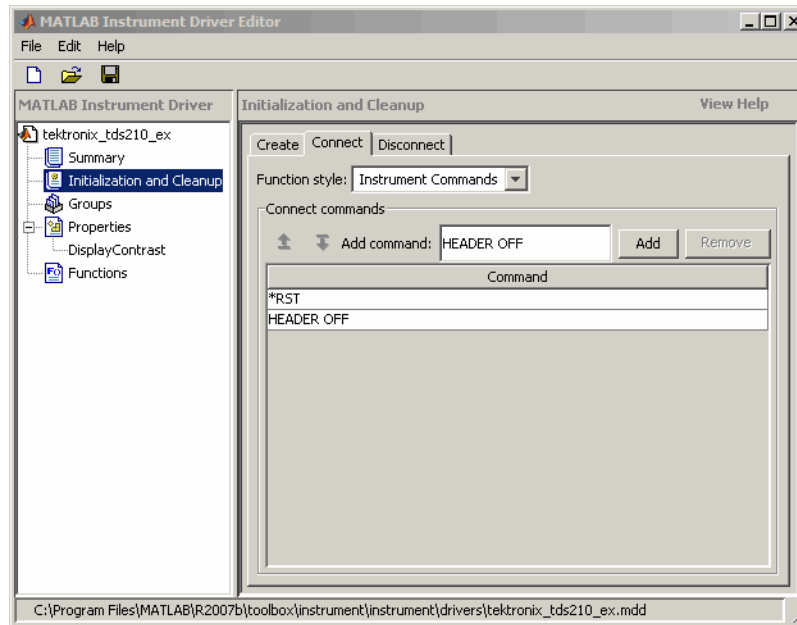
Verifying Connect Code

This procedure verifies the instrument commands defined in the connect code. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a Keithley GPIB board at board index 0.

1 From the MATLAB command line, 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 Verify the connect code by querying the Header state of the instrument.

```
query(g, 'Header?')
ans =
```

```
        0
```
4   Disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

Disconnect Code

By defining disconnect code, you can ensure that the instrument and the device object are returned to a known state after communication with the instrument is completed.

Disconnect code is evaluated before the device object's being disconnected from the instrument with the `disconnect` function. This allows the disconnect code to communicate with the instrument. Disconnect code can be defined as a series of instrument commands that will be written to the instrument or it can be defined as MATLAB software code.

Defining Disconnect Code

This example defines the disconnect code that ensures that the Tektronix TDS 210 oscilloscope is returned to a known state after communicating with the instrument using the device object.

1   From the MATLAB instrument driver editor, select the `Initialization and Cleanup` node in the tree.
2   Click the **Disconnect** tab and enter the MATLAB software code to execute when the device object is disconnected from the instrument.

   - Select `M-Code` from the **Function style** menu.

   - Define the MATLAB software code that will reset the instrument and configure the interface object's buffers to their default values.

     ```
     % Get the interface object.
     g = obj.Interface;

     % Reset the instrument to a known state.
     fprintf(g, '*RST');
     ```

3   Click the **Save** button.
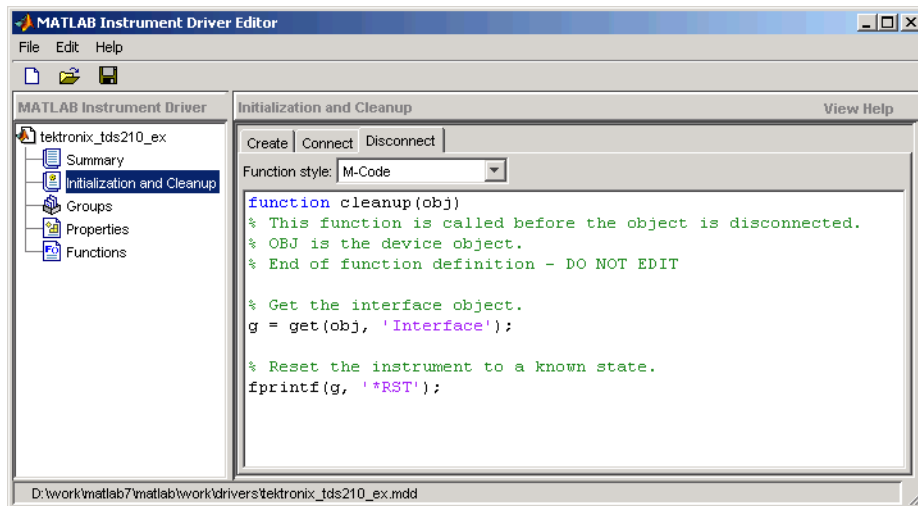
Verifying Disconnect Code

This procedure verifies the MATLAB software code defined in the disconnect code. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a 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

Properties: Overview

You can make the programming of instruments through device objects easier and more consistent by using properties. A property can be used to query or set an instrument setting or attribute. For example, an oscilloscope's trigger level may be controlled with a property called `TriggerLevel`, which you can read or control with the `get` or `set` function. Even if two different scopes have different trigger syntax, you can use the same property name, `TriggerLevel`, to control them, because each scope will have its own instrument driver.

Another advantage of properties is that you can define them with certain acceptable values (enumerated) or limits (bounded) that can be checked before the associated commands are sent to the instrument.

Property Components

The behavior of the property is defined by the following components.

Set Code

The `set` code defines the code that is executed when the property is configured with the `set` function. The `set` code can be defined as an instrument command that will be written to the instrument or it can be defined as MATLAB software code.

If the `set` code is MATLAB code, it can include any number of commands or MATLAB software code wrapped around instrument commands to provide additional processing or analysis.

If the `set` code is defined as an instrument command, then the command written to the instrument will be the instrument command concatenated with a space and the value specified in the call to `set`. For example, the `set` code for the `DisplayContrast`

property is defined as the instrument command `DISplay:CONTRast`. When the `set` function below is evaluated, the instrument command sent to the instrument will be `DISplay:CONTRast 54`.

```
set(obj, 'DisplayContrast', 54);
```

Get Code

The `get` code defines the code that is executed when the property value is queried with the `get` function. The `get` code can be defined as an instrument command that will be written to the instrument or it can be defined as MATLAB software code.

Note The code used for your property's `get` code and `set` code cannot include calls to the `fclose` or `fopen` functions on the interface object being used to access your instrument.

Accepted Property Values

You can define the values that the property can be set to so that only valid values are written to the instrument and an error would be returned before an invalid value could be written to the instrument.

- A property value can be defined as a double, a 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.

Read-Only Value	Description
Never	The property can be configured at all times with the `set` function.
While Open	The property can only be configured with the `set` function when the device object is not connected to the instrument. A device object is disconnected from the instrument with the `disconnect` function.
Always	The property cannot be configured with the `set` function.

Help Text

The help text provides information on the property. This information is returned with the `instrhelp` function.

```
instrhelp(obj, 'PropertyName')
```

Examples of Properties

This section includes several examples of creating, setting, and reading properties, with steps for verifying the behavior of these properties.

Creating a Double-Bounded Property

This example creates a property that will configure the Tektronix TDS 210 oscilloscope's LCD display contrast. The oscilloscope display can be configured to a value in the range [1 100]. In the MATLAB instrument driver editor,

1   Select the Properties node in the tree.
2   Enter the property name, DisplayContrast, in the **Name** text field and click the **Add** button. The new property's name, DisplayContrast, appears in the **Property Name** table.
3   Expand the Properties node in the tree to display all the defined properties.
4   Select the DisplayContrast node from the properties displayed in the tree.
5   Select the **Code** tab to define the set and get commands for the DisplayContrast property.

   •   Select Instrument Commands in the **Property style** field.

   •   Enter DISplay:CONTRast? in the **Get command** text field.

   •   Enter DISplay:CONTRast in the **Set command** text field.
6   Select the **Property Values** tab to define the allowed property values.

   •   Select Double in the **Data Type** field.

   •   Select Bounded in the **Constraint** field.

   •   Enter 1.0 in the **Minimum** field.

   •   Enter 100.0 in the **Maximum** field.
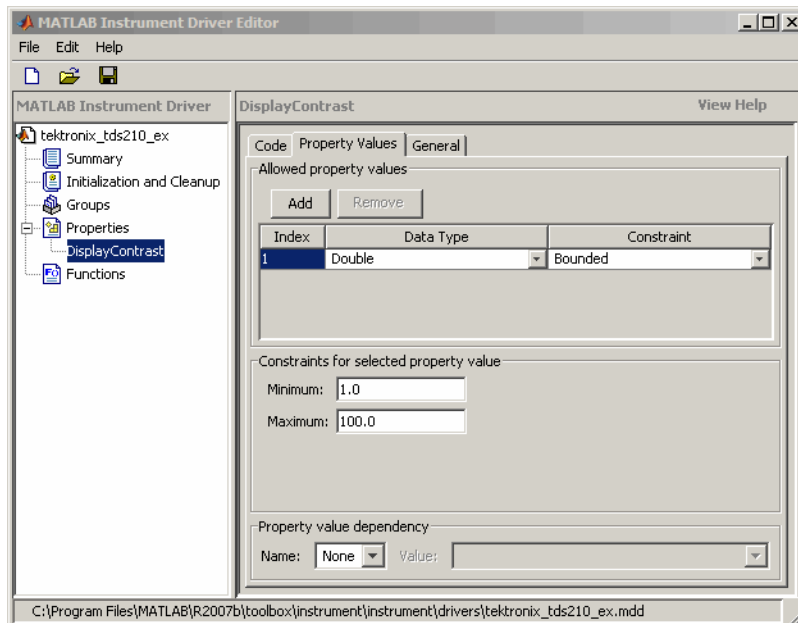
7 Select the **General** tab to finish defining the property behavior.

-   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.
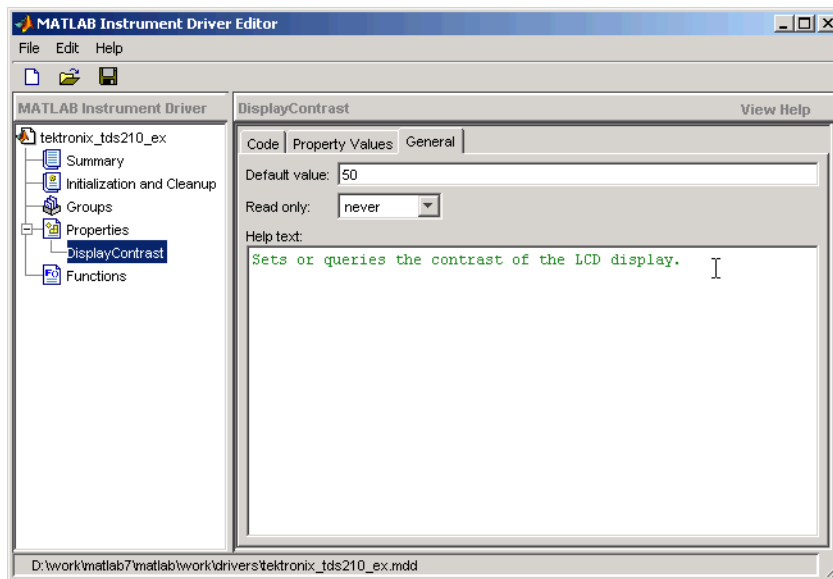
Verifying the Behavior of the Property

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.

```
g = gpib('keithley', 0, 2);
obj = icdevice('tektronix_tds210_ex.mdd', g);
```
2 View DisplayContrast property and its current value.

```
obj.DisplayContrast
ans =
    50
```
3 Calling set on the DisplayContrast property lists the values to which you can set the property.

```
set(obj, 'DisplayContrast')
[ 1.0 to 100.0 ]
```
4 Try setting the property to values inside and outside of the specified range.

```
obj.DisplayContrast = 17
obj.DisplayContrast
ans =
     17
obj.DisplayContrast = 120
??? Invalid value for DisplayContrast. Valid values: a value
between 1.0 and 100.0.
```

5 View the help you wrote.

```
instrhelp(obj,'DisplayContrast')
DISPLAYCONTRAST  [ 1.0 to 100.0 ]
Sets or queries the contrast of the LCD display.
```

6 List the DisplayContrast characteristics that you defined in the **Property Values** and **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 the MATLAB software, the tektronix_tds210_ex.mdd driver actually sends the DISplay:CONTRast? command to the instrument.

```
obj.DisplayContrast
ans =
     17
```

When you issue the set function in the MATLAB software, the tektronix_tds210_ex.mdd driver actually sends the DISplay:CONTRast 34 command to the instrument.

```
obj.DisplayContrast = 34)
```

8 Finally, disconnect from the instrument and delete the objects.

```
disconnect(obj)
```

```
delete([obj g])
```

Creating an Enumerated Property

This example creates a property that will select and display the Tektronix TDS 210 oscilloscope's cursor. The oscilloscope allows two types of cursor. It supports a horizontal cursor that measures the vertical units in volts, divisions, or decibels, and a vertical cursor that measures the horizontal units in time or frequency. In the MATLAB instrument driver editor,

1 Select the `Properties` node in the tree.
2 Enter the property name, `CursorType`, in the **Name** text field and click the **Add** button. The new property's name `CursorType` appears in the **Property Name** table.
3 Expand the `Properties` node to display all the defined properties.
4 Select the `CursorType` node from the properties displayed in the tree.
5 Select the **Code** tab to define the `set` and `get` commands for the `CursorType` property.

   • Select `Instrument Commands` in the **Property style** field.

   • Enter `CURSor:FUNCtion?` in the **Get Command** text field.

   • Enter `CURSor:FUNCtion` in the **Set Command** text field.

6 Select the **Property Values** tab to define the allowed property values.

   • Select `String` in the **Data Type** field.

   • Select `Enumeration` in the **Constraint** field.

   • Enter `none` in the **New property value** text field and click the **Add** button. Then enter `OFF` in the **Instrument Value** table field.

   • Similarly add the property value `voltage`, with instrument value `HBArs`.

   • Similarly add the property value `time`, with instrument value `VBArs`.
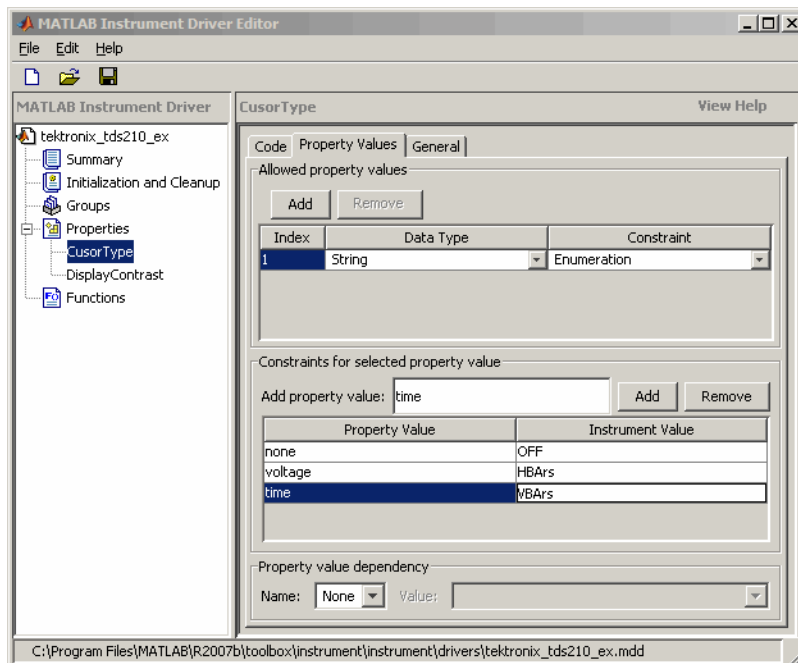
7 Select the **General** tab to finish defining the property behavior.

   • 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.

```
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.

```
get(obj)
```

Calling `get` on the `CursorType` property lists its current value.

```
obj.CursorType
ans =
    none
```

3 View acceptable values for the `CursorType` property. Calling `set` on the object lists all its settable properties.

```
set(obj)
```

Calling `set` on the `CursorType` property lists the values to which you can set the property.

```
set(obj, 'CursorType')
[ {none} | voltage | time ]
```

4 Try setting the property to valid and invalid values.

```
obj.CursorType = 'voltage';
obj.CursorType
ans =
    voltage
obj.CursorType = 'horizontal'
??? 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
```

18-25

```
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 the MATLAB software, the `tektronix_tds210_ex.mdd` driver actually sends the `CURSor:FUNCtion VBArs` command to the instrument.

```
obj.CursorType = 'time'
```

When you issue the `get` function in the MATLAB software, the `tektronix_tds210_ex.mdd` driver actually sends the `CURSor:FUNCtion?` command to the instrument.

```
obj.CursorType
ans =
time
```

8 Finally disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

A MATLAB Code Style Property

This example creates a property that will return the difference between two cursors of the Tektronix TDS 210 oscilloscope. The oscilloscope allows two types of cursor. It supports a horizontal cursor that measures the vertical units in volts, divisions, or decibels, and a vertical cursor that measures the horizontal units in time or frequency. The previous example created a property, CursorType, that selects and displays the oscilloscope's cursor. In the MATLAB instrument driver editor,

1 Select the `Properties` node in the tree.
2 Enter the property name, `CursorDelta`, in the **New Property** text field and click **Add**. The new property's name, `CursorDelta`, appears in the **Property Name** table.
3 Expand the `Properties` node to display all the defined properties.
4 Select the `CursorDelta` node from the properties displayed in the tree.

5 Select the **Code** tab to define the set and get commands for the `CursorDelta` property.

- Select `M-Code` in the **Property style** field.

- Since the `CursorDelta` property is read-only, no MATLAB software code will be added to the **MATLAB Set Function** text field.

- The following MATLAB software code is added to the **MATLAB Get Function** text field.

```
% Extract the interface object.
interface = obj.Interface;

% Determine the type of cursor being displayed.
type = obj.CursorType

% Based on the cursor type, query the instrument.
switch (type)
case 'none'
    propertyValue = 0;
case 'voltage'
    propertyValue = query(interface, 'CURSor:HBArs:DELTa?');
    propertyValue = str2double(propertyValue);
case 'time'
    propertyValue = query(interface, 'CURSor:VBArs:DELTa?');
    propertyValue = str2double(propertyValue);
end
```
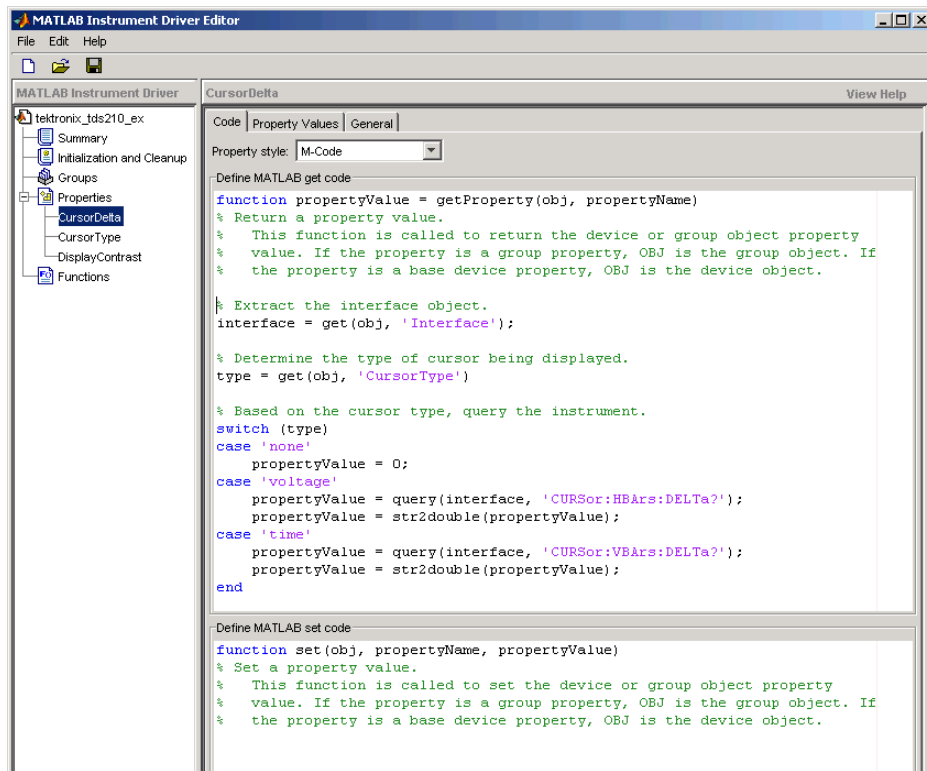
6   Select the **Property Values** tab to define the allowed property values.

   • Select `Double` in the **Data Type** field.

   • Select `None` in the **Constraint** field.

7   Select the **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.

```
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.

```
get(obj)
```

3 View the `CursorDelta` property's current value.

```
obj.CursorDelta
ans =
     0
```

4 Calling `set` on the object lists all its settable properties. Note that as a read-only property, `CursorDelta` is not listed in the output.

```
set(obj)
```

5 Setting the property to a value results in an error message.

```
obj.CursorDelta = 4)
??? Changing the 'CursorDelta' property of device objects is not
allowed.
```

6 View the help you wrote.

```
instrhelp(obj,'CursorDelta')
CURSORDELTA  (double)  (read only)
Returns the difference between the two cursors.
```

7 List the `CursorDelta` characteristics that you defined in the **Property Values** and **General** tabs.

```
info = propinfo(obj,'CursorDelta')
info =
                 Type: 'double'
           Constraint: 'none'
      ConstraintValue: []
         DefaultValue: 0
             ReadOnly: 'always'
```

```
        InterfaceSpecific: 1
```

8 Connect to your instrument to verify the `get` code.

```
connect(obj)
```

When you issue the `get` function in the MATLAB software, the `tektronix_tds210_ex.mdd` driver actually executes the MATLAB software code that was specified.

```
obj.CursorDelta
ans =
    1.6000
```

9 Finally, disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

Functions

Understanding Functions

Functions allow you to call the instrument to perform some task or tasks, which may return results as text data or numeric data. The function may involve a single command to the instrument, or a sequence of instrument commands. A function may include the MATLAB software code to determine what commands are sent to the instrument or to perform analysis on data returned from the instrument. For example, a function may request that a meter run its self-calibration, returning the status as a result. Another function may read a meter's scaling, request a measurement, adjust the measured data according to the scale reading, and then return the result.

Function Components

The behavior of the function is defined by the components described below.

MATLAB Code

The MATLAB code defines the code that is executed when the function is evaluated with the `invoke` function. The code can be defined as an instrument command that will be written to the instrument or it can be defined as the MATLAB software code.

If the code is defined as an instrument command, the instrument command can be defined to take an input argument. All occurrences of `<input argument name>` in the instrument command are substituted with the input value passed to the `invoke` function. For example, if a function is defined with an input argument, `start`, and the instrument command is defined as `Data:Start <start>`, and a start value of `10` is passed to the `invoke` function, the command `Data:Start 10` is written to the instrument.

If the code is defined as an instrument command, the instrument command can also be defined to return an output argument. The output argument can be returned as numeric data or as text data.

If the code is defined as the MATLAB software code, you can determine which commands are sent to the instrument, and the data results from the instrument can be manipulated, adjusted, or analyzed as needed.

Note The code used for your function's MATLAB software code cannot include calls to the `fclose` or `fopen` functions on the interface object being used to access your instrument.

Help Text

The help text provides information on the function.

Examples of Functions

This section includes several examples of functions, and steps to verify the behavior of these functions.

Simple Function

This example creates a function that will cause the Tektronix TDS 210 oscilloscope to adjust its vertical, horizontal and trigger controls to display a stable waveform. In the MATLAB instrument driver editor,

1 Select the `Functions` node in the tree.
2 Enter the function name, `autoset`, in the **Add function** text field and click the **Add** button. The new function's name, `autoset`, appears in the **Function Name** table.
3 Expand the `Functions` node to display all the defined functions.
4 Select the `autoset` node from the functions displayed in the tree.
5 Select the **Code** tab to define commands executed for this function.

  - Select `Instrument Commands` in the **Function style** field.

  - In the **Function commands** pane, enter `AUTOSet EXECute` in the **Add command** field and click the **Add** button.
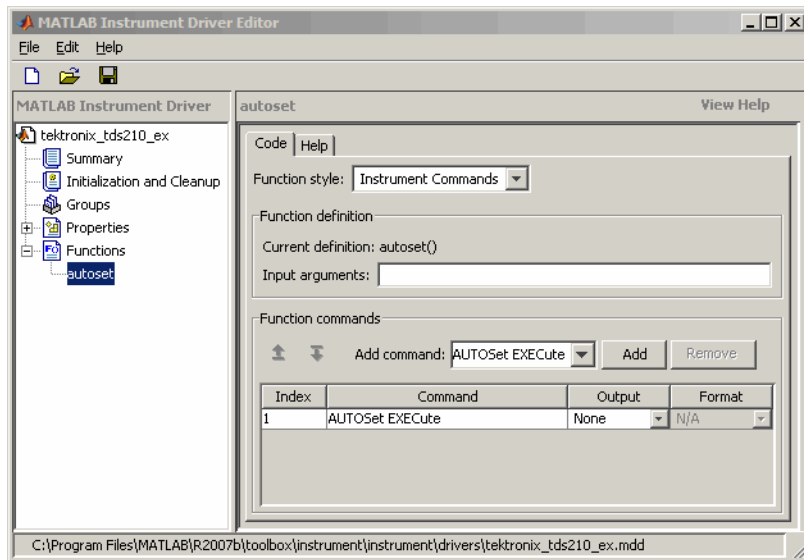
6   Select the **Help** tab to define the help text for this function.

     •   In the **Help text** field, enter INVOKE(OBJ, 'autoset') causes the oscilloscope to adjust its vertical, horizontal, and trigger controls to display a stable waveform.

7   Click the **Save** button.

Verifying the Behavior of the Function

This procedure verifies the behavior of the function. In this example, the driver name is tektronix_tds210_ex.mdd. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a 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
end             instrfindall    ne
eq              instrhelp       obj2mfile

Driver specific methods for class icdevice:

autoset
```

3 View the help you wrote.

```
instrhelp(obj, 'autoset')
INVOKE(OBJ, 'autoset') causes the oscilloscope to adjust its
vertical, horizontal, and trigger controls to display a stable
waveform.
```

4 Using the controls on the instrument, set the scope so that its display is unstable. For example, set the trigger level outside the waveform range so that the waveform scrolls across the display.

5 Connect to your instrument and execute the function. Observe how the display of the waveform stabilizes.

```
connect(obj)
invoke(obj, 'autoset')
```

6 Disconnect from your instrument and delete the object.

```
disconnect(obj)
delete([obj g])
```

Function with Instrument Commands that Use Input and Output Arguments

This example creates a function that configures which waveform will be transferred from the Tektronix TDS 210 oscilloscope, and configures the waveform's starting and ending data points. In the MATLAB instrument driver editor,

1 Select the Functions node in the tree.

2 Enter the function name, `configureWaveform`, in the **Add function** text field and click the **Add** button. The new function's name, `configureWaveform`, appears in the **Function Name** pane.

3 Expand the `Functions` node to display all the defined functions.

4 Select the `configureWaveform` node from the functions displayed in the tree.

5 Select the **Code** tab to define commands executed for this function.

- Select `Instrument Commands` in the **Function style** field.

- Enter the input arguments `source, start, stop` in the **Input arguments** field.

- Enter `Data:Source <source>` in the **Add command** field and click the **Add** button. In the table, select an **Output** type of `None` and a **Format** type of `N/A`.

- Similarly, add the command: `Data:Source?` with `ASCII` **Output** and `text` **Format**.

- Similarly, add the command: `Data:Start <start>` with `NONE` **Output** and `N/A` **Format**.

- Similarly, add the command: `Data:Start?` with `ASCII` **Output** and `numeric` **Format**.

- Similarly, add the command: `Data:Stop <stop>` with `NONE` **Output** and `N/A` **Format**.

- Similarly, add the command: `Data:Stop?` with `ASCII` **Output** and `numeric` **Format**.
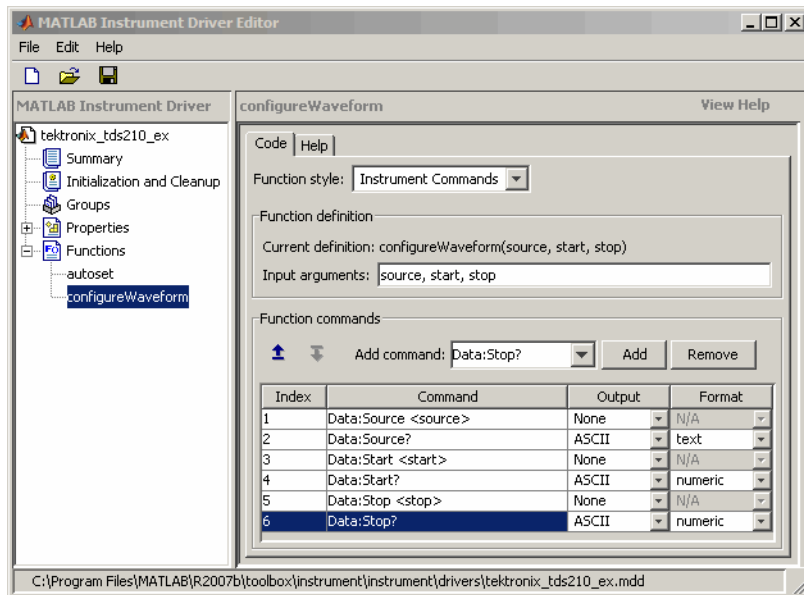
6   Select the **Help** tab to define the help text for this function.

   •   In the **Help text** field, enter `[SOURCE, START, STOP] = INVOKE(OBJ,`
       `'configureWaveform', SOURCE, START, STOP) configures the`
       `waveform that will be transferred from the oscilloscope`.

7   Click the **Save** button.

Verifying the Behavior of the Function

This procedure verifies the behavior of the function. In this example, the driver name is
`tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope
at primary address 2 is done via a 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
end              instrfindall   ne
eq               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])
```

MATLAB Code Style Function

This example creates a function that will transfer and scale the waveform from the Tektronix TDS 210 oscilloscope. In the MATLAB instrument driver editor,

1   Select the `Functions` node in the tree.

2   Enter the function name, `getWaveform`, in the **Add function** text field and click the **Add** button. The new function's name, `getWaveform`, appears in the **Function Name** table.

3   Expand the `Functions` node to display all the defined functions.

4   Select the `getWaveform` node from the functions displayed in the tree.

5   Select the **Code** tab to define commands executed for this function.

- Select `M-Code` in the **Function style** field.

- Update the function line in the **Define MATLAB function** text field to include an output argument.

  ```
  function yout = getWaveform(obj)
  ```

- Add the following MATLAB software code to the **Define MATLAB function** text field. (The instrument may require a short pause before any statements that read a waveform, to allow its completion of the data collection.)

  ```
  % Get the interface object.
  g = obj.Interface;

  % Configure the format of the data transferred.
  fprintf(g, 'Data:Encdg SRIBinary');
  fprintf(g, 'Data:Width 1');

  % Determine which waveform is being transferred.
  source = query(g, 'Data:Source?');

  % Read the waveform.
  fprintf(g, 'Curve?');
  ```

```
ydata = binblockread(g, 'int8');

% Read the trailing terminating character.
fscanf(g);

% Scale the data.
fprintf(g, ['WFMPre:' source ':Yoff?']);
yoffset = fscanf(g, '%g');

fprintf(g, ['WFMPre:' source ':YMult?']);
ymult = fscanf(g, '%g');

yout = (ydata*ymult) + yoffset;
```
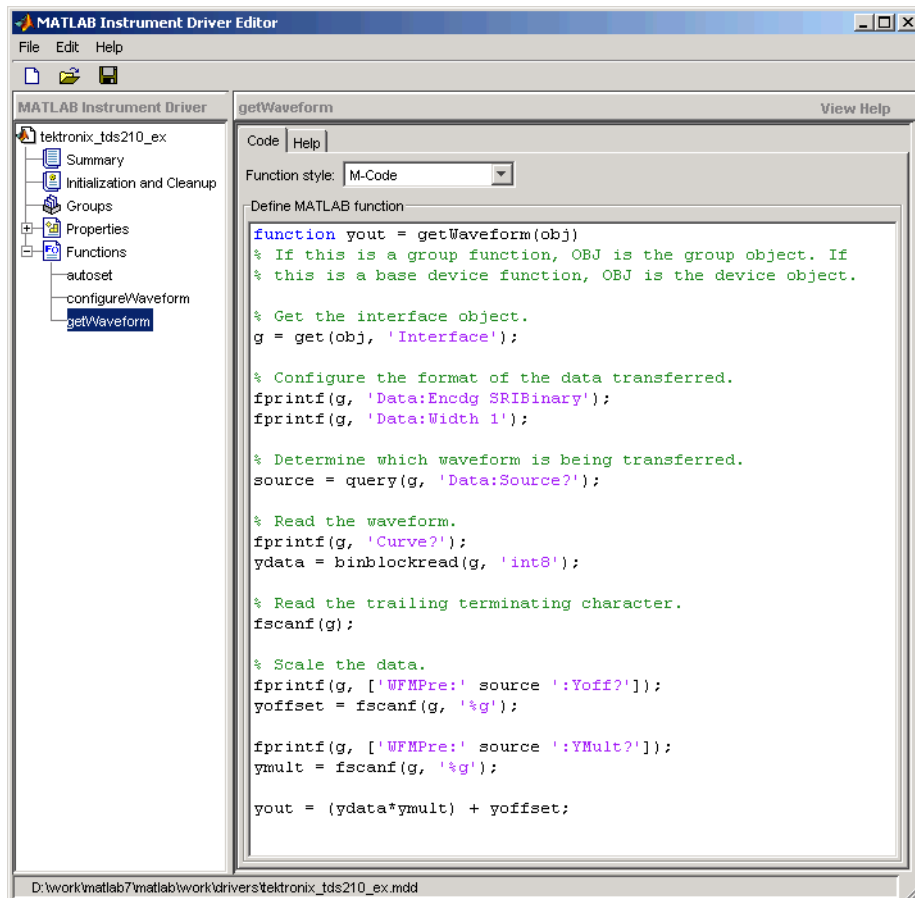
6 Click the **Help** tab to define the help text for this function.

- In the **Help text** field, enter `DATA = INVOKE(OBJ, 'getWaveform')`
  `transfers and scales the waveform from the oscilloscope`.

7 Click the **Save** button.

Verifying the Behavior of the Function

This procedure verifies the behavior of the function. In this example, the driver name is
`tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope

at primary address 2 is done via a 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
end               instrfindall     ne
eq                instrhelp        obj2mfile

Driver specific methods for class icdevice:

autoset configureWaveform getWaveform
```

3 View the help you wrote.

```
instrhelp(obj, 'getWaveform')
DATA = INVOKE(OBJ, 'getWaveform') transfers and scales the
waveform from the oscilloscope.
```

4 Connect to your instrument and execute the function.

```
connect(obj)
```

Configure the waveform that is going to be transferred.

```
invoke(obj, 'configureWaveform', 'CH1', 1, 500);
```

Transfer the waveform.

```
data = invoke(obj, 'getWaveform');
```

18-41

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 18-43
"Examples of Groups" on page 18-44

Group Components

A group may be used to set or query the same property on several elements, or to query several related properties, at the same time. For example, all input channels on an oscilloscope can be scaled to the same value with a single command; or all current measurement setups can be retrieved and viewed at the same time.

A group consists of one or more group objects. The objects in the group share a set of properties and functions. Using these properties and functions you can control the features of the instrument represented by the group. In order for the group objects to control the instrument correctly, the group must define a selection command for the group and an identification string for each object in the group.

Selection Command

The selection command is an instrument command that configures the instrument to use the capability or physical component represented by the current group object. Note, the instrument might not have a selection command.

Identification String

The identification string identifies an object in the group. The number of identification strings listed by the group defines the number of objects in the group. The identification string can be substituted into the instrument commands written to the instrument.

When a group object instrument command is written to the instrument, the following steps occur:

1 The selection command for the group is determined by the driver.
2 The identification string for the group object is determined by the driver.
3 If the selection command contains the string `<ID>`, it is replaced with the identification string.

4   The selection command is written to the instrument. If empty, nothing is written to the instrument.

5   If the instrument command contains the string `<ID>`, it is replaced with the identification string.

6   The instrument command is written to the instrument.

Examples of Groups

This section includes several examples of groups, with steps to verify the code.

Creating a One-Element Group

This example combines the trigger capabilities of the Tektronix TDS 210 oscilloscope into a trigger group. The oscilloscope allows the trigger source and slope settings to be configured. In the MATLAB instrument driver editor,

1   Select the `Groups` node in the tree.

2   Enter the group name, `Trigger`, in the **Add Group** text field and click **Add**.

3   Expand the `Groups` node to display all the defined groups.

4   Select the `Trigger` node in the tree.

5   Select the **Definition** tab.

Since the oscilloscope has only one trigger, there is not a command that will select the current trigger. The **Selection command** text field will remain empty.
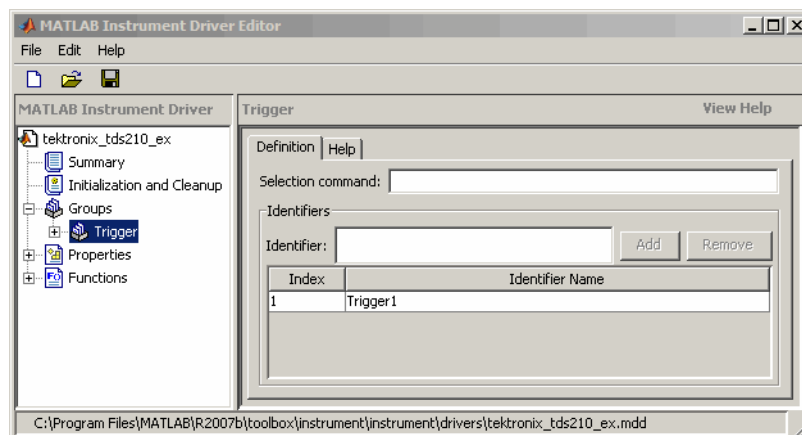
6   Select the **Help** tab to finish defining the group behavior.

In the **Help text** field, enter `Trigger is a trigger group. The trigger group object contains properties that configure and query the oscilloscope's triggering capabilities.`

7 Click the **Save** button.

Verifying the Group Behavior

This procedure verifies the group information defined. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a 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 group you created. Note that the `HwName` property is the group object identification string.

```
get(obj)
obj.Trigger
HwIndex:   HwName:     Type:           Name:
1          Trigger1    scope-trigger   Trigger1
```

3 View the help.

```
instrhelp(obj, 'Trigger')
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.

```
delete([obj g])
```

Defining the Group Object Properties for a One-Element Group

This example defines the properties for the `Trigger` group object created in the previous example. The Tektronix TDS 210 oscilloscope can trigger from CH1 or CH2 when the data has a rising or falling slope.

First, the properties `Source` and `Slope` are added to the trigger group object. In the MATLAB instrument driver editor,

1 Expand the `Trigger` group node to display the group object's properties and functions.
2 Select the `Properties` node to define the `Trigger` group object properties.
3 Enter the property name `Source` in the **Add property** text field and click the **Add** button
4 Enter the property name `Slope` in the **Add property** text field and click the **Add** button.
5 Expand the `Properties` node to display the group object's properties.

Next, define the behavior of the `Source` property:

1 Select the `Source` node in the tree.
2 Select the **Code** tab to define the `set` and `get` commands for the `Source` property.

   - Select `Instrument Commands` in the **Property style** field.

   - Enter `TRIGger:MAIn:EDGE:SOUrce?` in the **Get command** text field.

   - Enter `TRIGger:MAIn:EDGE:SOUrce` in the **Set command** text field.
3 Select the **Property Values** tab to define the allowed property values.

   - Select `String` in the **Data Type** field.

   - Select `Enumeration` in the **Constraint** field.

   - Enter CH1 in the **Add property value** text field and click the **Add** button. Then enter CH1 in the **Instrument Value** table field.

   - Similarly, add the enumeration: CH2, CH2.
4 Select the **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`.
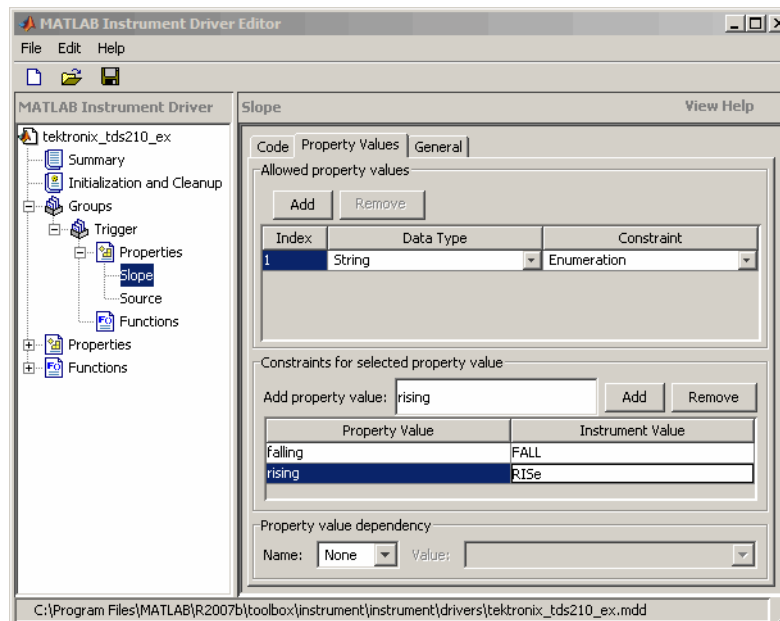
4 Select the **General** tab to finish defining the property behavior.

- 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 = obj.Trigger
    HwIndex:    HwName:     Type:           Name:
    1           Trigger1    scope-trigger   Trigger1
```

3   Access specific properties to list its current value.

```
t.Source
ans =
    CH1

t.Slope
ans =
    falling
```

4   Calling `set` on a specific property lists the values to which you can set the property.

```
set(t, 'Source')
[ {CH1} | CH2 ]

set(t, 'Slope')
[ {falling} | rising ]
```

5   Try setting the property to valid and invalid values.

```
t.Source = CH2;

t.Slope ='rising'

t.'Source
ans =
    CH2

t.Slope
ans =
    'rising'
```

```
t.Source = 'CH3'
??? The 'CH3' enumerated value is invalid.

t.Slope = 'steady'
??? The 'steady' enumerated value is invalid.
```

6 View the help you wrote.

```
instrhelp(t, 'Source')
SOURCE [ {CH1} | CH2 ]
Specifies the source for the main edge trigger.

instrhelp(t, 'Slope')
SLOPE [ {falling} | rising ]
Specifies a rising or falling slope for the main edge trigger.
```

7 List the group object characteristics that you defined in the **Property Values** and **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
```

8 Connect to your instrument to verify the `set` and `get` code.

```
connect(obj)
```

Note When you issue the `get` function on the `Source` property for the trigger object, the `textronix_tds210_ex.mdd` driver actually sends the `TRIGger:MAIn:EDGE:SOUrce?` command to the instrument.

```
t.Source
ans =
CH1
```

> **Note** When you issue the `set` function on the `Slope` property for the trigger object, the `textronix_tds210_ex.mdd` driver actually sends the `TRIGger:MAIn:EDGE:SLOpe RISe` command to the instrument.

```
t.Slope = 'rising'
```

9 Disconnect from your instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

Creating a Four-Element Group

This example combines the measurement capabilities of the Tektronix TDS 210 oscilloscope into a measurement group. The oscilloscope allows four measurements to be taken at a time. In the MATLAB instrument driver editor,

1 Select the `Groups` node in the tree.
2 Enter the group name, `Measurement`, in the **Add group** text field and click **Add**.
3 Expand the `Groups` node to display all the defined groups.
4 Select the `Measurement` node in the tree.
5 Select the **Definition** tab.

- The oscilloscope does not define an instrument command that will define the measurement that is currently being calculated. The **Selection command** text field will remain empty.

- In the **Identifier Name** listing, change `Measurement1` to `Meas1` to define the identification string for the first measurement group object in the group.

- Enter the identifiers `Meas2`, `Meas3`, and `Meas4` for the remaining measurement group objects by typing each in the **Identifier** text field and clicking **Add** after each.

6   Select the **Help** tab to finish defining the group behavior.

    • In the **Help text** field, enter `Measurement is an array of measurement group objects. A measurement group object contains properties related to each supported measurement on the oscilloscope.`

7   Click the **Save** button.

Verifying the Group Behavior

This procedure verifies the group information defined. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a 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 group you created. Note that the `HwName` property is the group object `get(obj)`.

```
obj.Measurement

HwIndex:    HwName:    Type:               Name:
1           Meas1      scope-measurement   Measurement1
2           Meas2      scope-measurement   Measurement2
3           Meas3      scope-measurement   Measurement3
4           Meas4      scope-measurement   Measurement4
```

3 View the help.

```
instrhelp(obj, 'Measurement')
MEASUREMENT (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.

```
delete([obj g])
```

Defining the Group Object Properties for a Four-Element Group

This example defines the properties for the `Measurement` group object created in the previous example. The Tektronix TDS 210 oscilloscope can calculate the frequency, mean, period, peak to peak value, root mean square, rise time, fall time, positive pulse width, or negative pulse width of the waveform of Channel 1 or Channel 2.

First, the properties `MeasurementType`, `Source`, `Value`, and `Units` will be added to the `Measurement` group object.

1 Expand the `Measurement` group node to display the group object's properties and methods.

2 Select the `Properties` node to define the `Measurement` group object properties.

3 Enter the property name `MeasurementType` in the **Add property** text field and click the **Add** button.

4 Enter the property name `Source` in the **Add property** text field and click the **Add** button.

5 Enter the property name `Value` in the **Add property** text field and click the **Add** button.

6 Enter the property name `Units` in the **Add property** text field and click the **Add** button.

7 Expand the `Properties` node to display the group object's properties.
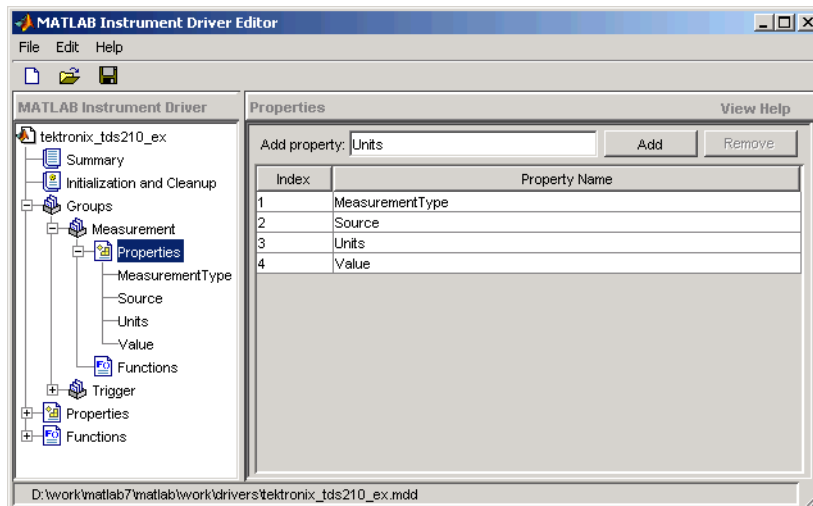
Next, define the behavior of the MeasurementType property:

1 Select the MeasurementType node in the tree.

2 Select the **Code** tab to define the set and get commands for the MeasurementType property.

- Select Instrument Commands in the **Property style** field.

- Enter Measurement:<ID>:Type? in the **Get command** text field.

- Enter Measurement:<ID>:Type in the **Set command** text field.

3 Select the **Property Values** tab to define the allowed property values.

- Select String in the **Data Type** field.

- Select Enumeration in the **Constraint** field.

- Enter frequency in the **Add property value** text field and click the **Add** button. Then enter FREQuency in the **Instrument Value** table field.

- Add the enumeration: mean, MEAN.

- Add the enumeration: period, PERIod.

- Add the enumeration: pk2pk, PK2pk.

- Add the enumeration: rms, CRMs.

- Add the enumeration: riseTime, RISe.

- Add the enumeration: `fallTime`, `FALL`.
- Add the enumeration: `posWidth`, `PWIdth`.
- Add the enumeration: `negWidth`, `NWIdth`.
- Add the enumeration: `none`, `NONE`.
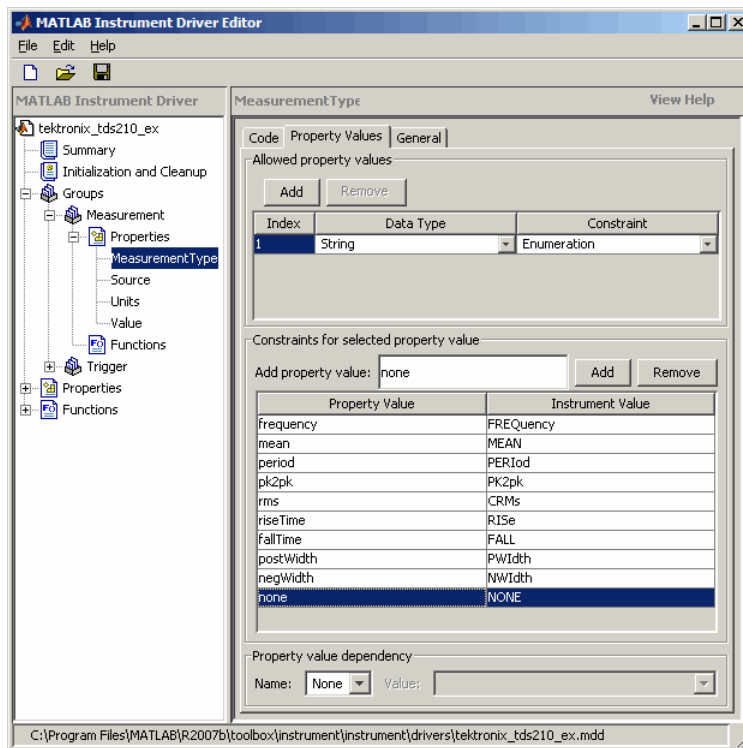
4 Select the **General** tab to finish defining the property behavior.

- Enter `none` in the **Default value** text field.
- Select `never` in the **Read only** field.
- In the **Help text** field, enter `Specifies the 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 `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 `Returns the measurement value`.

5 Click the **Save** button.

Verifying the Properties of the Group Object in the MATLAB software

This procedure verifies the properties of the measurement group object. In this example, the driver name is `tektronix_tds210_ex.mdd`. Communication with the Tektronix TDS 210 oscilloscope at primary address 2 is done via a 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 = obj.Measurement
```

HwIndex:	HwName:	Type:	Name:
1	Meas1	scope-measurement	Measurement1
2	Meas2	scope-measurement	Measurement2
3	Meas3	scope-measurement	Measurement3
4	Meas4	scope-measurement	Measurement4

3 View the current values for the properties of the first group object. Calling `get` on the object lists all its properties.

```
m(1)
```

4 Calling `get` on a specific property lists its current value.

```
m(1).MeasurementType

ans =
none

m(1).Source

ans =
CH1

m(1).Units

ans =
volts

m(1).Value

ans =
'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')
[ {CH1} | CH2 ]
```

6 Try setting the property to valid and invalid values.

```
m(1).Source = 'CH2'
m(1).Source
ans =
CH2
m(1).Source = 'CH5'
??? The 'CH5' enumerated value is invalid.
```

7 View the help you wrote.

```
instrhelp(m(1), 'Value')
VALUE (double) (read only)
Returns the measurement value.
```

18-57

8 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
```

9 Connect to your instrument to verify the `set` and `get` code.

```
connect(obj)
```

Note When you issue the `get` function on the `MeasurementType` property for the first measurement object in the group, the `textronix_tds210_ex.mdd` driver actually sends the `Measurement:Meas1:Type?` command to the instrument.

```
m(1).MeasurementType
ans =
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.

```
m(2).Source = 'CH2';
```

10 Disconnect from your instrument and delete the objects.

```
disconnect(obj)
delete([obj g])
```

Using Existing Drivers

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 18-5 for information on

· Defining summary information

· Defining initialization and cleanup code

· Creating a new property

· Creating a new function

· Creating a new group

Importing VXI*plug&play* and IVI Drivers

The MATLAB Instrument Driver Editor can import a VXI*plug&play* or IVI driver. You can evaluate or set the driver's functions and properties, and the modified driver can be saved for further use:

1 Open the MATLAB Instrument Driver Editor with `midedit`.
2 Click the **File** menu, and select **Import**.

The **Import Driver** dialog box appears, showing the installed VXI*plug&play* and IVI drivers.
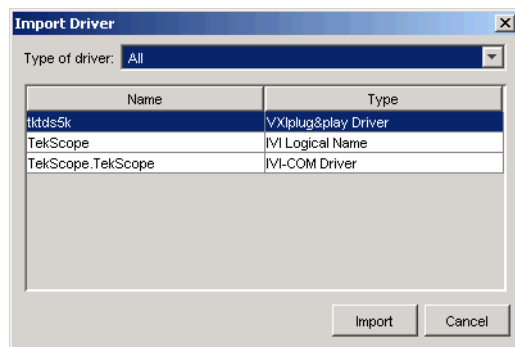
3 Select the desired driver and click **Import**.

The MATLAB Instrument Driver Editor creates a MATLAB instrument driver based on the properties and/or functions in the original VXI*plug&play* or IVI driver. The editor displays the new driver's summary information, groups, properties, and functions.

With the MATLAB Instrument Driver Editor, you can

- Create, delete, modify, or rename properties, functions, or groups
- Add code around instrument commands for analysis
- Add create, connect, and disconnect code
- Save the driver as a separate MATLAB VXI*plug&play* instrument driver or MATLAB IVI instrument driver

19

Using the Instrument Driver Testing Tool

This chapter describes how to use the Instrument Driver Testing Tool to verify the functionality of your instrument drivers.

Instrument Driver Testing Tool Overview

Functionality

This section provides an overview of the MATLAB Instrument Driver Testing Tool and examples showing its capabilities and usage.

The MATLAB Instrument Driver Testing Tool provides a graphical environment for creating a test to verify the functionality of a MATLAB instrument driver.

The MATLAB Instrument Driver Testing Tool provides a way to do the following:

- Verify property behavior.
- Verify function behavior.
- Save the test as a test file, a MATLAB code, or driver function.
- Export the test results to MATLAB workspace, figure window, MAT-file, or the MATLAB Variables editor.
- Save test results as an HTML page.

Drivers

You can use the MATLAB Instrument Driver Testing Tool to test any MATLAB instrument driver, which include:

- MATLAB interface drivers
- MATLAB VXI*plug&play* drivers
- MATLAB IVI drivers

MATLAB VXI*plug&play* drivers and MATLAB IVI drivers can be created from VXI*plug&play* and IVI drivers, respectively, using the MATLAB Instrument Driver Editor or the `makemid` function.

Test Structure

The driver test structure is composed of setup information and test steps.

Setup

When setting up or initializing the test, you provide a test name and description, identify the driver to test, define the interface to the instrument, and set the test preferences. This information remains unchanged throughout the execution of the test, and applies to every step.

Test Steps

The executable portion of the test is divided into any number of test steps. A test step can perform one of four verifications:

- Set property — Verify that the set command or set code of a single device object or group object property in the driver does not error, and that the driver supports the defined range for the property value. You can use one value or all supported values for the property. You may also use invalid property values to check the driver's response.
- Get property — Verify the reading of a single device object or group object property from the driver.
- Properties sweep — Verify several properties in a single step.
- Function — Verify the execution of a driver function.

After configuring your test steps, you can execute the steps individually, or run a complete test that executes all the steps in the test.

Starting

You start the MATLAB Instrument Driver Testing Tool by typing the MATLAB command

```
midtest
```

This opens the tool without any test file loaded.

You may specify a test file (usually created in an earlier session of the tool) when you start the tool so that it opens up with a test file already loaded.

```
midtest('MyDriverTestfile')
```

Note: MIDTEST and the Instrument Driver Testing Tool are unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

Example

For the examples in this chapter, you will create a test for the Tektronix TDS210 oscilloscope driver that you created in "MATLAB Instrument Driver Editor Overview" on page 18-2.

You will create each kind of step in your test: set property, get property, sweep properties, and function.

Setting Up Your Test

In this section...

Test File

You can specify a test file to load when you start `midtest`, open a test file after the MATLAB Instrument Driver Testing Tool is already up, or create a new test. You may find it convenient to keep the driver and test file together in the same directory. For easy use in the MATLAB Command Window, you can put that directory in the MATLAB path with the `addpath` command.

Note: MIDTEST and the Instrument Driver Testing Tool are unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

Providing a Name and Description

The **Name** field allows a one-line text definition for your test. This name appears in the header of the test results in the Output Window.

The **Description** field allows a full definition of the text with as much descriptive text as you need.

Specifying the Driver

In the **Driver** text field, you specify the driver to be tested. This is any MATLAB instrument driver, usually with the `.mdd` extension. Enter the full path to the driver, or click **Browse** to navigate to the driver's directory.

Specifying an Interface

You specify the interface with the instrument for the testing of the driver. The instrument object type may be GPIB, VISA, TCPIP, UDP, or serial port. Depending on the type you choose, the **New Object Creation** dialog box prompts you for further configuration information.

The tool then creates a device object based on interface and driver.

Setting Test Preferences

The **Test Preferences** dialog box allows you to set certain behaviors of the tool when running a test.

Run Mode

This specifies whether the test runs all the steps or only one step in the test.

Fail Action

This specifies what happens if a step within the test fails. The test may stop after the failed step or continue, with or without resetting the instrument.

No-error String

This field specifies the expected string returned from the instrument *when there is no error*. If you indicate that a step passes when no error is returned from the instrument, the tool compares the string returned from the instrument via the geterror function, to the string given here in the Preferences dialog box. If the strings match, then the tool assumes there is no error from the instrument.

Number of Values to Test

A double-precision property can be tested using all supported values. You can request this when testing it as a single step, or the tool does it automatically when the property is tested as part of a property sweep step. This field specifies how many values are tested for such a property.

The number of values includes the defined minimum and maximum for the property, and integer values equally spaced between these limits.

If your property requires noninteger values for testing, then create a separate test step for that property instead of including it in a sweep.

Setting Up a Driver Test

This example identifies the driver to be tested, and defines global setup information for the test. You will be testing the driver created in the examples of "MATLAB Instrument Driver Editor Overview" on page 18-2.

1   Open the MATLAB Instrument Driver Testing Tool from the command line with the command `midtest`.
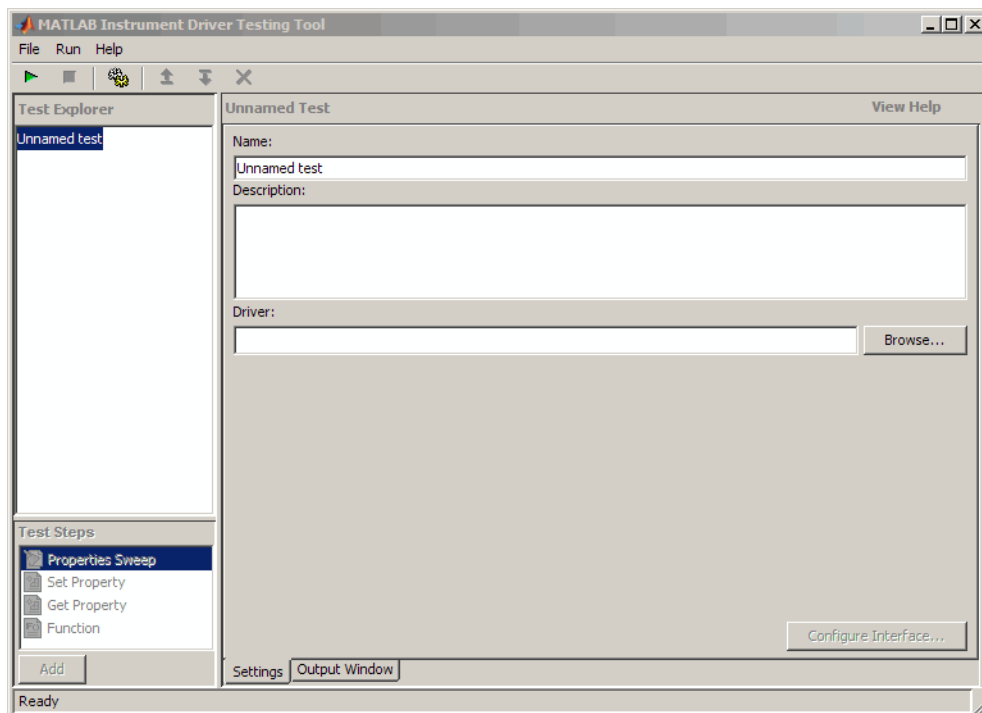
2   In the **Name** text field, enter `TDS 210 Driver Sample Test`.
3   In the **Description** text field, enter `A test to check some of the properties and functions of the TDS 210 oscilloscope driver`.
4   In the **Driver** field, enter the name of the driver you created in "MATLAB Instrument Driver Editor Overview" on page 18-2. The text field will display the whole pathname, with the driver file `tektronix_tds210_ex.mdd`.
5   Click the **Create** button to create an instrument interface.
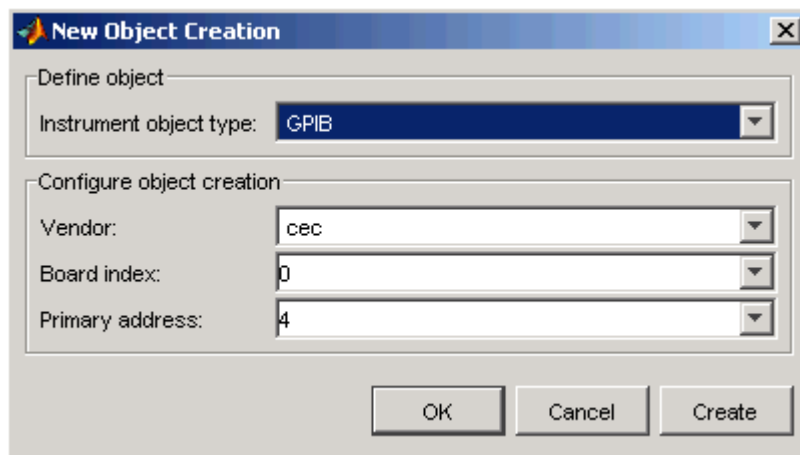
6   In the **New Object Creation** dialog box,

   a   Select your **Instrument object type**, **Vendor**, **Board index**, and **Primary address** of your instrument.

   The example illustrations in this chapter use a CEC GPIB board with index 0 and the instrument at address 4. Your configuration may be different.

   b   Click **OK**.

7   Click the **File** menu and select **Test Preferences**.

8   In the **Test Preferences** dialog box,

   a   For **Select run mode**, click **Run all steps**.

   b   For **Select fail action**, click **Continue test**.

   c   For **Message returned from instrument when no error occurred**, enter " ". (This is an empty string in double quotes.)

   d   For **Number of values to test for double properties**, enter 5.

   e   Click **OK**.
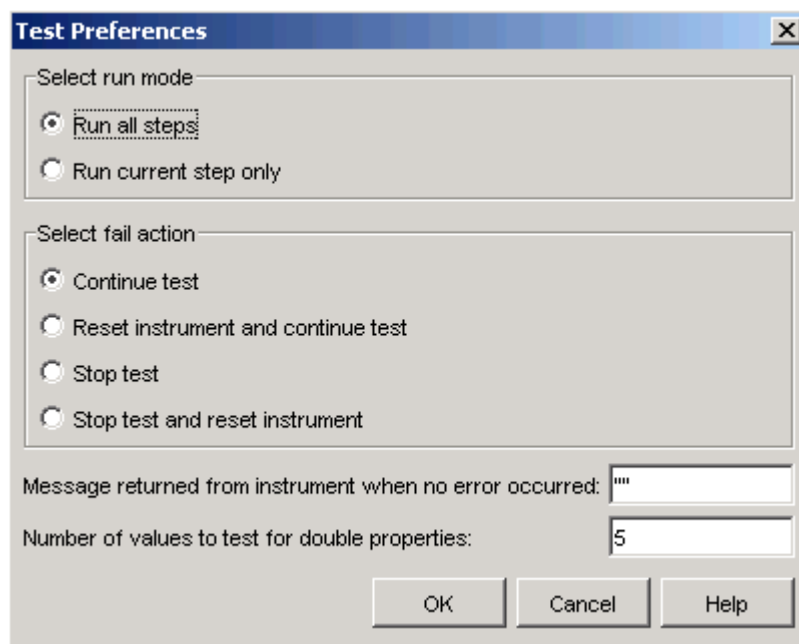
The MATLAB Instrument Driver Testing Tool now displays all your setup information.
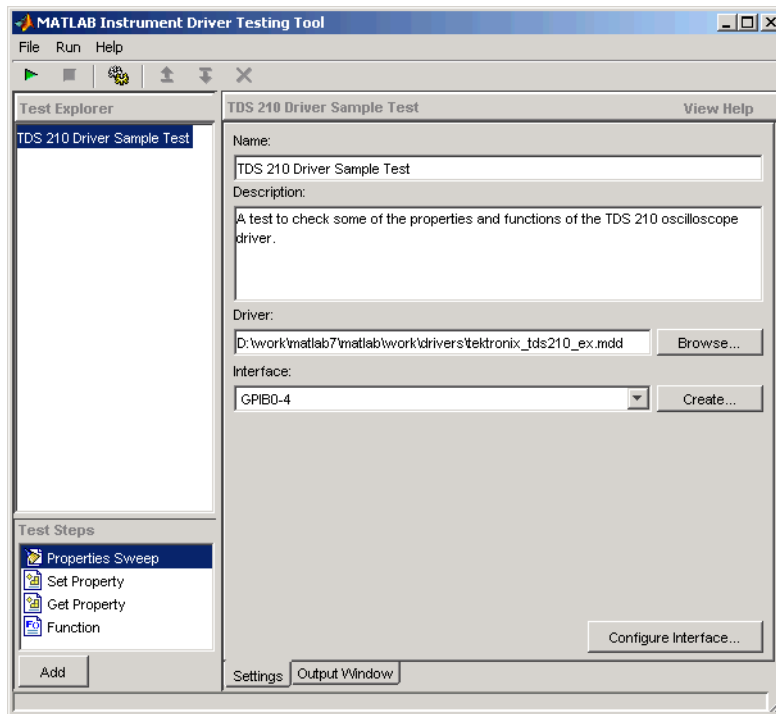
9 Click **File** and select **Save**. Enter `tektronix_tds210_ex_test` as the filename for your test. The tool automatically adds the `.xml` file extension.

Defining Test Steps

In this section...

Test Step: Set Property

You use a set property test step to verify a driver's set code or set command for a property. You provide a name for the step, select the driver property to test and the values to test it with, and define the conditions for the step's passing.

Settings	Dscription
Name	You provide a name for each test step. The name appears in the **Test Explorer** tree as well as in the results output.
Property to Test	A set property step can test only one property. You choose the property from the Property to Test list. Additional properties can be tested with additional steps, or with a sweep step.
Object(s) to Test	A property may be defined for the instrument or for a group object. If you are testing a group object property, you select which object you want tested in the Object(s) to Test list.
Define the Values to Test	If the property is has enumerated values, you can select one of the defined values, all of the supported values, or some other value. If the property's value is a double-precision number, you can select a value within its defined range, all supported values, or some other value. For a double, you set the number of values tested for all supported values in the Preferences dialog box (see "Number of Values to Test" on page 19-6).
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

Settings	Dscription
	If you select more than one of these conditions, then both conditions must be met for the step to pass. If no boxes are selected, the test will pass.

Creating a Test Step: Set Property

1 Click the `Set Property` option in the **Test Steps** list box.
2 Click the **Add** button.
3 In the **Name** field, enter `Set Display Contrast`.
4 In the **Property to test** list, select `DisplayContrast`.
5 For **Define the value(s) to test**, select `All supported values`.
6 For **Select when this step passes**,

  • Select **If no MATLAB software or instrument error occurs**.

  • Select **If current value matches configured value**.
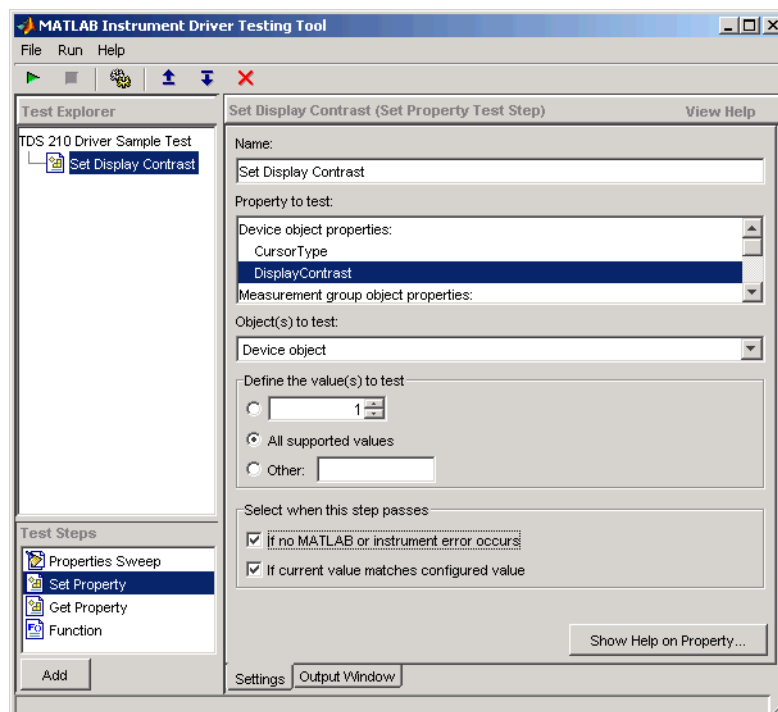
7   Click **File** and select **Save**.

Running a Test Step to Set a Property

You can run an individual test step to verify its behavior:

1   Select `Set Display Contrast` in the **Test Explorer** tree.
2   With the cursor on the selected name, right-click to bring up the context menu.
3   In the context menu, select **Run this step only**.
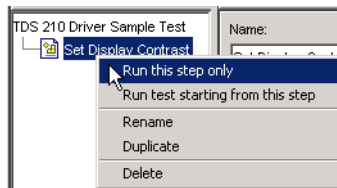
You may want to repeat this step as you observe the oscilloscope display. The test sets the display contrast to five different values: lowest acceptable value (1%), highest acceptable value (100%), and three approximately equally spaced integer values between these limits.

The tool automatically displays the **Output Window** with the test results.
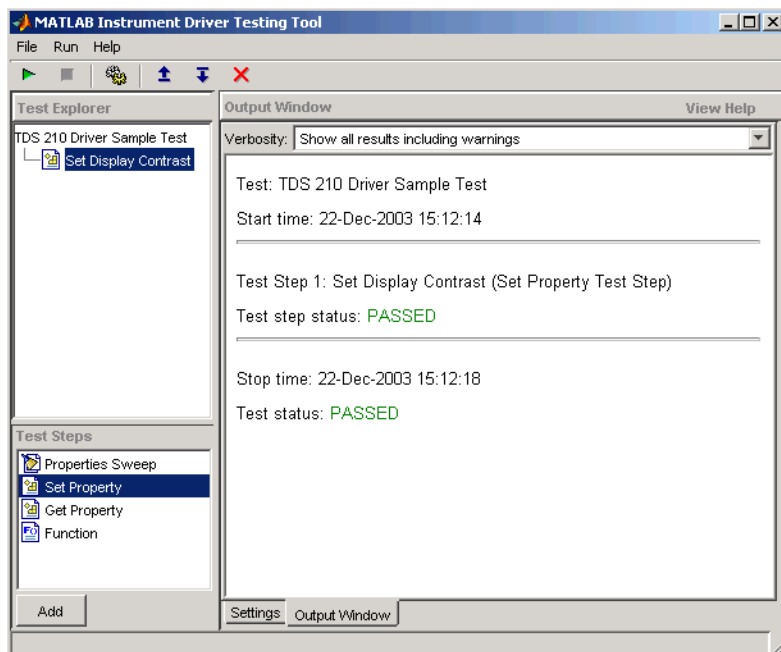
This test step passed because, for each of the five display contrast settings, the tool read back a value that was equal to the configured value.

Test Step: Get Property

You use a get property test step to verify a driver's ability to read a property. You provide a name for the step, select the driver property to test, and define the conditions for the step's passing.

Settings

The settings for the get property step are the same as for a "Test Step: Set Property" on page 19-11, 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 19-27, after the test step has executed.

Creating a Test Step: Get Property

1 Click the `Get Property` option in the **Test Step** field.
2 Click the **Add** button.
3 In the **Name** field, enter `Getting Display Contrast`.
4 In the **Property to test** list, select `DisplayContrast`.
5 In the **Output argument** field, enter `DispContr`.
6 For **Select when this step passes**,

- Unselect the box for **If no MATLAB software or instrument error occurs**.

- Select **If property value is**, and enter a value of `80`.

  This value is chosen to generate a failure. If this step follows the previous step in the example, the display contrast is still set at `100`. If this step is run by itself, the display contrast is set to `50` by the `*RST` command that is executed as part of your connect code for the driver.
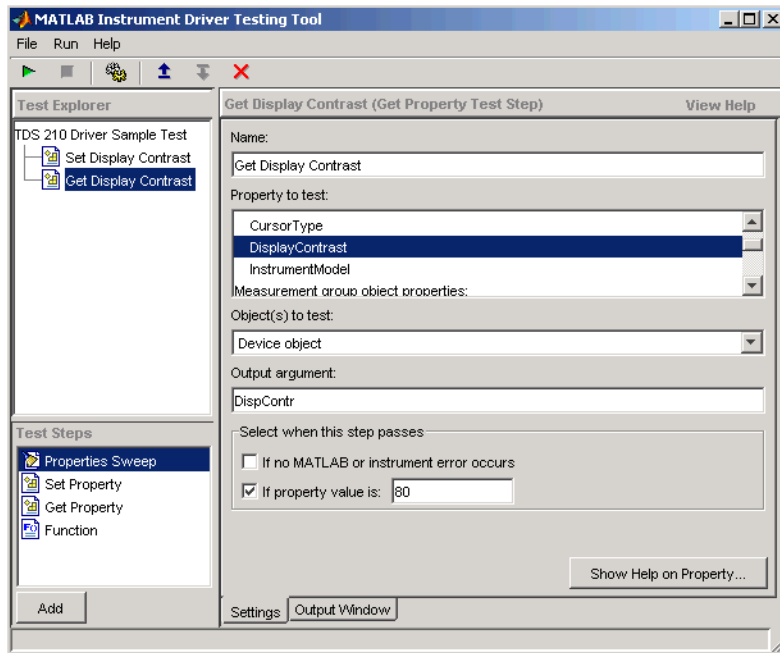
7 Click **File** and select **Save**.

Running a Test Step to Get a Property

You run the individual test step to verify its behavior.

1 Select `Get Display Contrast` in the **Test Explorer** tree.
2 With the cursor on the selected name, click the right mouse button to bring up the context menu.
3 In the context menu, select **Run this step only**.

Note that the test fails, reading a value of 50 while expecting a value of 80.
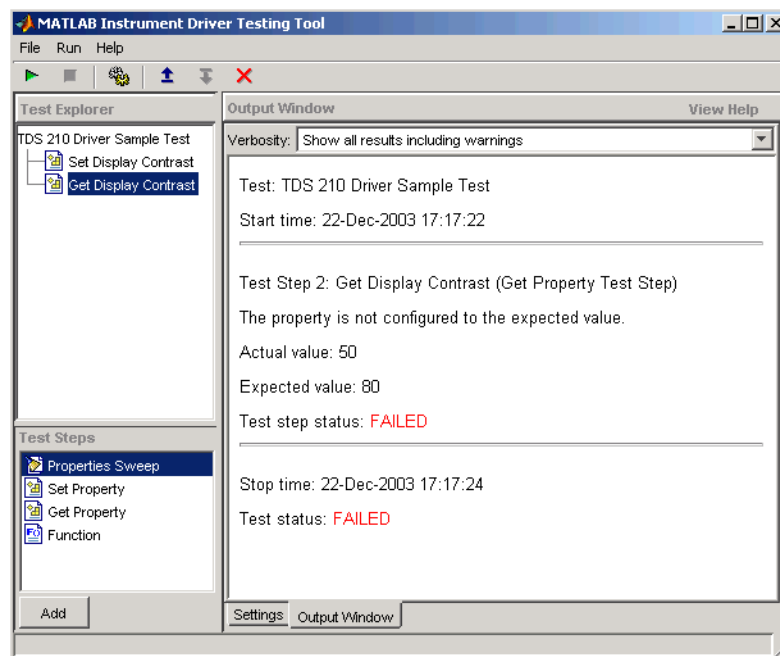
Test Step: Properties Sweep

A properties sweep step allows you to test several properties in a single step. All selected properties are tested for all supported values. (In the case of properties with double-precision values, you determine the "Number of Values to Test" on page 19-6, 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.

Creating a Sweep Step to Test All Properties

1   Click the `Properties Sweep` option in the **Test Step** field.
2   Click the **Add** button.
3   In the **Name** field, enter `All Properties Sweep`.
4   For **Select the properties to test**, click **Select All**.
5   In the Select the group object(s) field,

  • For the **Measurement** group, select `All Measurement group objects`.

  • For the **Trigger** group, select `All Trigger group objects`.
6   For **Select when this step passes**,

  • Select **If no MATLAB software or instrument error occurs**, and

  • Select **If current value matches configured value**
7   Click **File** and select **Save**.
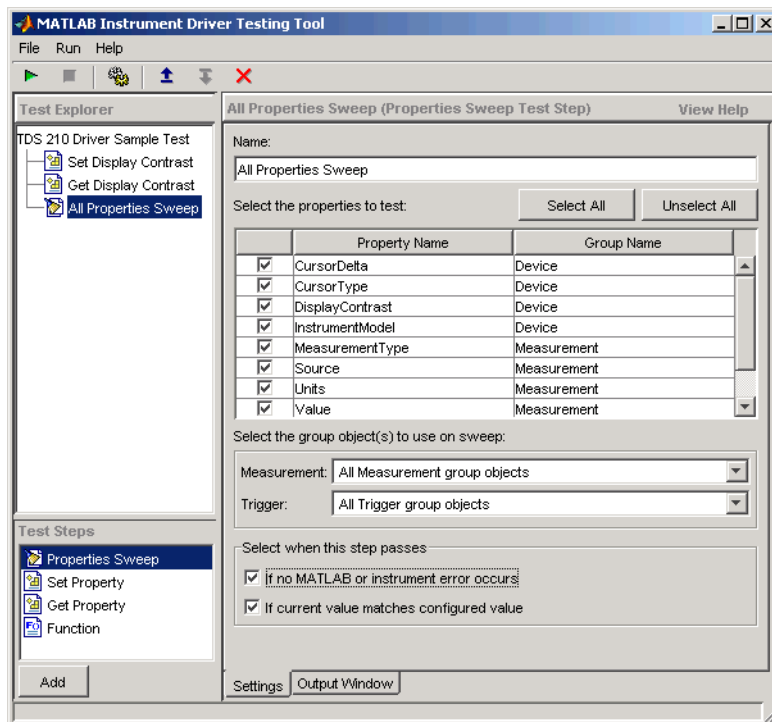
Running a Sweep Step to Test All Properties

You run the sweep test step to verify its behavior.

1 Select All Properties Sweep in the **Test Explorer** tree.
2 With the cursor on the selected name, click the right mouse button to bring up the context menu.
3 In the context menu, select **Run this step only**.

The **Output Window** is updated as each property in the sweep is tested. Note that the entire sweep is only one step in the overall test.
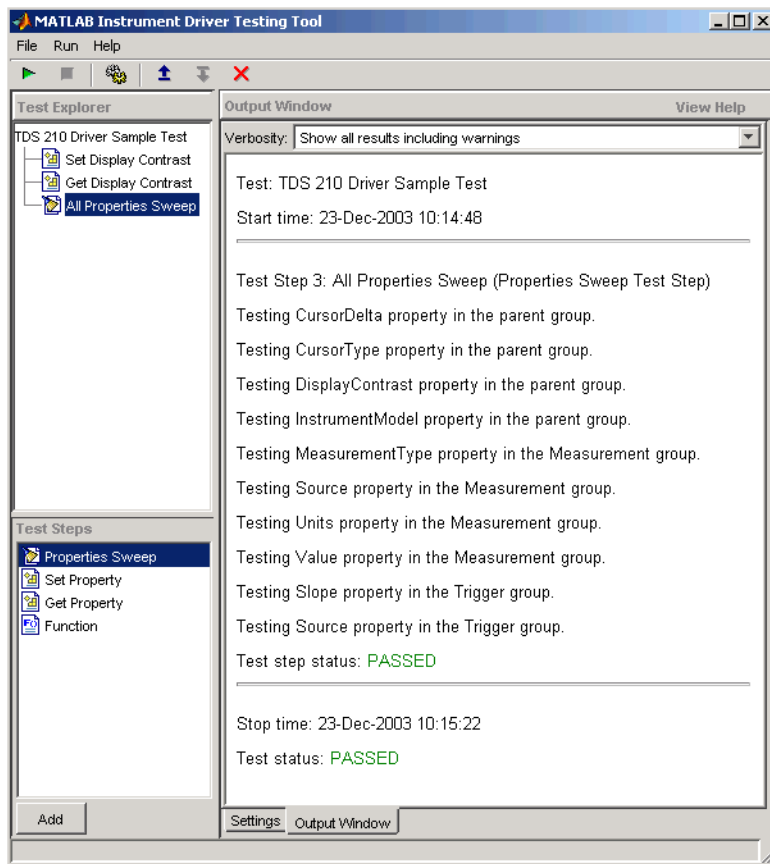
Test Step: Function

A function test step sends a function call to the instrument. You select the function called, the input data and output arguments (if required), and the conditions for passing.

Settings

Name

You provide a name for each test step. The name appears in the **Test Explorer** tree as well as in the results output.

Function to test

A function step can test only one function. You choose the function from the **Function to test** list. Additional functions can be tested with additional steps.

Function definition

The tool displays below the selected function what the call command for the function looks like. This helps you when deciding what input and output arguments to supply.

Input argument(s) and Output argument(s)

You provide input arguments as a comma-separated list of data, strings, 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 19-27 after the test step has executed.

Select when this step passes

The step passes when any of three conditions is met:

- If no instrument or MATLAB software error occurs as a result of attempting to execute the function
- If the returned output arguments match expected values
- If the output of a specified function is true

If you select more than one of these conditions, then all selected conditions must be met for the step to pass. If no boxes are selected, the test will pass.

Creating a Test Step: Function

1 Click the `Function` option in the **Test Step** field.
2 Click the **Add** button.
3 In the **Name** field, enter `Config Waveform`.
4 In the **Function to test** list, select `configureWaveform`.
5 In the **Input argument(s)** field, type `'CH1', 1, 3000`.
6 In the **Output argument(s)** field, type `Channel, StartAdr, StopAdr`.
7 For **Select when this step passes**,

- Select **If no MATLAB software or instrument error occurs**.
- Select **If output arguments are**, and enter in its field `'CH1', 1, 2500`.

> - Unselect **If output of function ... is true**.

8   Click **File** and select **Save**.

Note that you set the input argument for the stop address to `3000`, but you set the
expected value for its output argument, `StopAdr`, to `2500`. This is because the maximum
address of the oscilloscope is `2500`. If you attempt to exceed that value, the oscilloscope
address is set to the maximum.
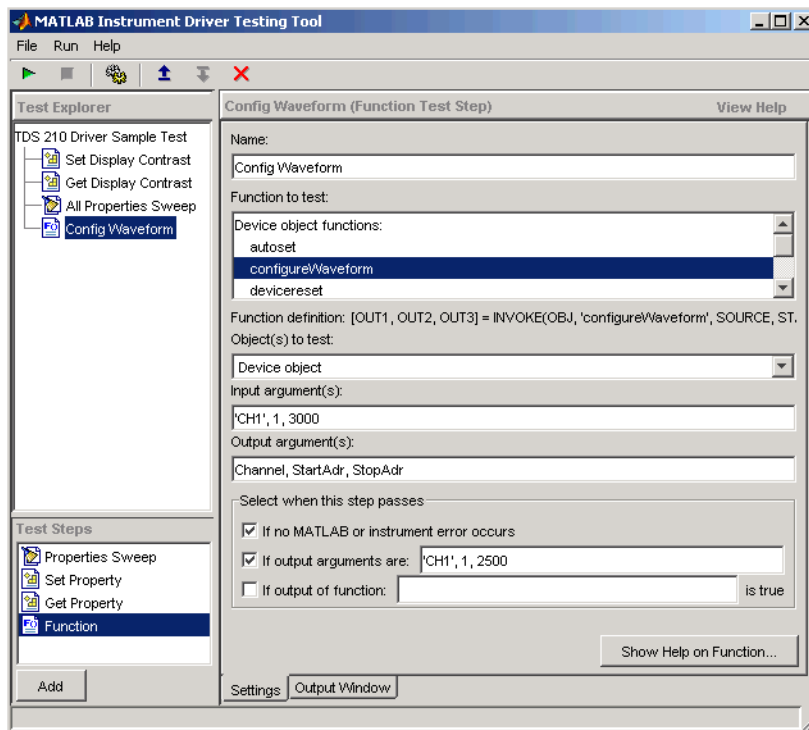
Running a Test Step to Test a Function

You can run an individual test step to verify its behavior

1   Select `Config Waveform` in the **Test Explorer** tree.
2   With the cursor on the selected name, click the right mouse button to bring up the
context menu.
3   In the context menu, select **Run this step only**.
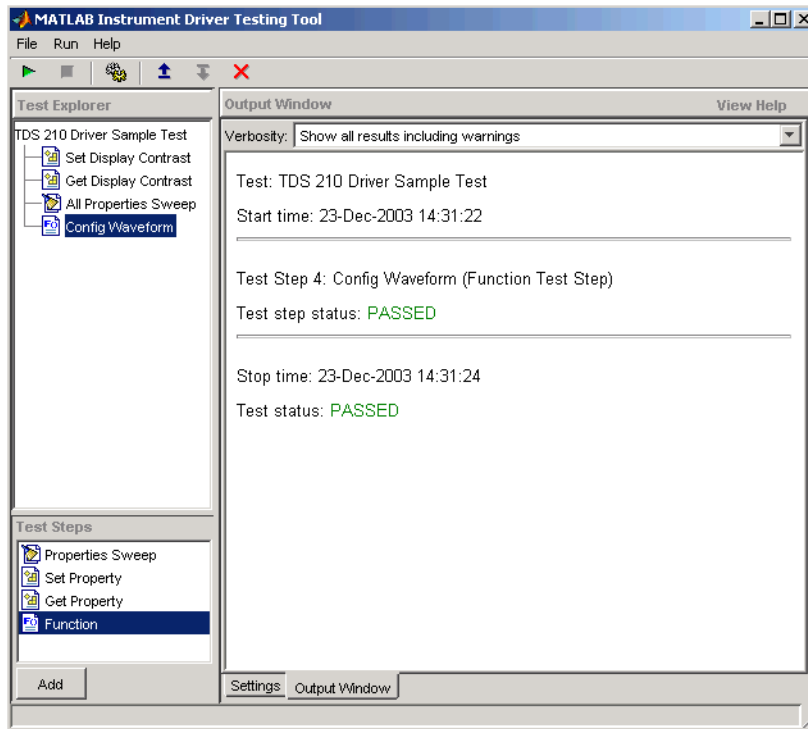
Saving Your Test

In this section...
"Saving the Test as MATLAB Code" on page 19-23
"Saving the Test as a Driver Function" on page 19-23

Saving the Test as MATLAB Code

In the preceding examples of this chapter, you have been saving the test file after creating each step. The test file is saved in XML format. Here are some other save options.

You save the test file as MATLAB code by clicking the **File** menu and selecting **Save Test as M-Code**.

You can execute the test by calling this file from the MATLAB Command Window.

For example, you can save the test file you created in this chapter as `tektronix_tds210_ex_test.m`. Then you execute the test from the MATLAB Command Window by typing

`tektronix_tds210_ex_test`

The test results are displayed in the MATLAB Command Window.

Saving the Test as a Driver Function

You save your test as a driver function by clicking the **File** menu and selecting **Save Test as Driver Function**.
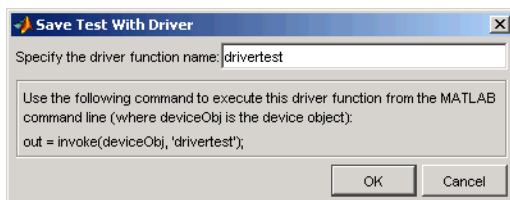
When you enter a name for the driver test function, the `invoke` command at the bottom of the dialog box reflects that name. You use that `invoke` command to execute the driver function from the MATLAB Command Window or in a file.

Creating a Driver Test Function

1 Click the **File** menu and select **Save Test as Driver Function**.
2 Enter `drivertest` in the **Specify the driver function name** field.
3 Click **OK**.

A function called `drivertest` is created and saved as part of the instrument driver file. You can open the driver file in the MATLAB Instrument Driver Editor tool (`midedit`) to verify that the `drivertest` function is included.

Calling a Driver Test Function from the MATLAB Command Window

Now that the test function is included in the driver, you access it with the `invoke` command from MATLAB.

In the MATLAB Command Window,

1 Create an interface object.

```
g = gpib('cec',0,4)
```
2 Create a device object, specifying the driver with the `drivertest` function saved in it.

```
obj = icdevice('tektronix_tds210_ex.mdd',g)
```
3 Connect to the device.

```
connect(obj)
```
4 Execute the driver test.

```
out = invoke(obj, 'drivertest')
```
5 When the test is complete, disconnect from the instrument and delete the objects.

```
disconnect(obj)
delete ([g obj])
```

Testing and Results

In this section...
"Running All Steps" on page 19-25
"Partial Testing" on page 19-27
"Exporting Results" on page 19-27
"Saving Results" on page 19-28

Running All Steps

So far in this chapter, you have only run individual test steps after each was created.

When you run the entire test, all the test steps run in the order listed in the **Test Explorer** tree. Using the mouse, you may drag the nodes of the tree to alter their sequence.

The **Output Window** displays the results of each step, along with a final result of the complete test.

Running a Complete Test

1   Select `Get Display Contrast` in the **Test Explorer** tree.
2   In the **Select when this step passes** field, change the **If property value is** entry from `80` to `100`.

   Earlier you entered a value of `80` to illustrate what a failure looks like. The display contrast is left at `100` from the `Set Display Contrast` test step, so that is what you will test for in the next step.
3   Click **File** and select **Save**.
4   Click the green arrow button to start a test run.
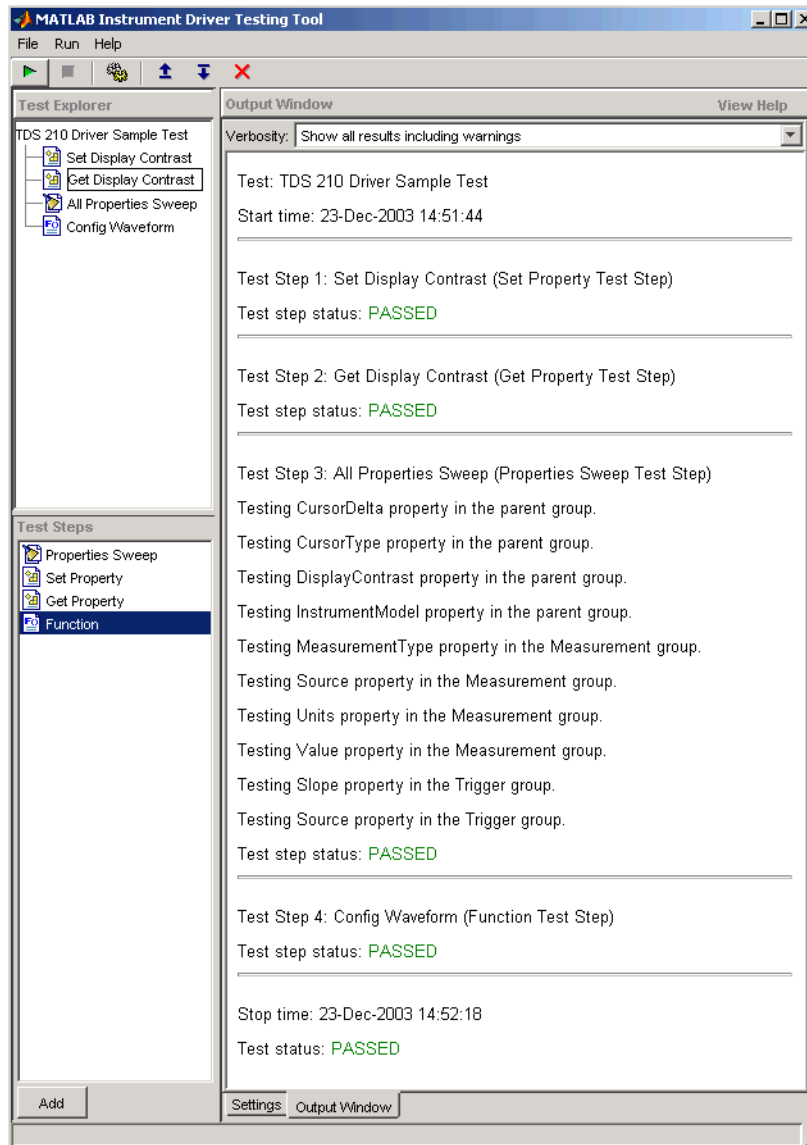
Partial Testing

Using the context menu in the **Test Explorer** tree, you can run a partial test of either an individual test step, or from the chosen test step through the end of the test.

Exporting Results

You can export the test results to many locations:

- MATLAB workspace
- MATLAB figure window
- MAT-file
- MATLAB Variables editor

The results you can export are those assigned to output variables in the settings for a test step.

Exporting Test Results to the MATLAB Workspace

1 Click the **File** menu and select **Export Test Results**.
2 In the **Test Results Exporter** dialog box, select `MATLAB Workspace` as the **Data destination**.

   By default, all the variables are selected. You may unselect any.
3 Click the **Export** button.
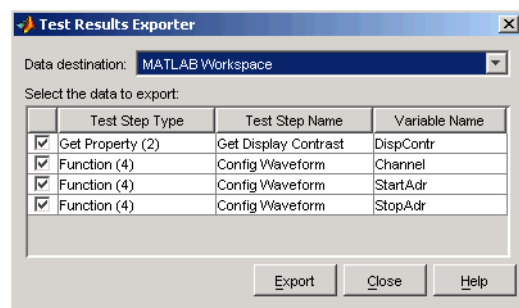
The variables are now available in the MATLAB workspace, with values that were established by the test run.
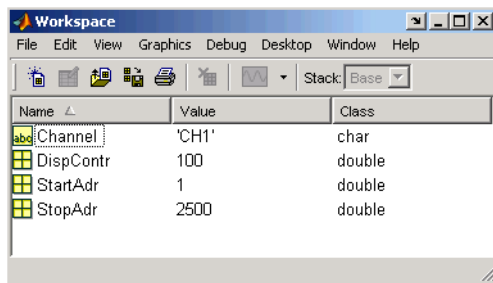
Saving Results

You save your test results in an HTML file by clicking the **File** menu and selecting **Save Test Results**. The format of the results in this file reflects their appearance in the tester tool's **Output Window**.

Instrument Control Toolbox Troubleshooting

How to Use This Troubleshooting Guide

If you have trouble connecting to or communicating with an instrument, try the suggestions in this guide.

The first thing to check is that your instrument is supported with the toolbox. See "Is My Hardware Supported?" on page 20-3 for information about supported interfaces and supported hardware.

For connection and communication issues using a specific interface, see the section about that interface. For example, if you are having trouble using an instrument over the Bluetooth interface, refer to the Bluetooth section. Each interface section covers platform support, interface requirements, and troubleshooting tips and procedures for that interface.

If you are using VISA with another interface, try reading the sections for both interfaces. For example, if you are using VISA with UDP, try reading the sections on both VISA ("IVI, VISA, and the Quick Control Interfaces" topic) and UDP.

If you are having trouble with deployment or use of the MATLAB Compiler™, see "Deploying Standalone Applications with Instrument Control Toolbox" on page 20-49.

If you need to contact MathWorks Technical Support, read "Contact MathWorks and Use the instrsupport Function" on page 20-53 first. The `instrsupport` function runs diagnostics and provides useful information that may help solve your problem.

Is My Hardware Supported?

In this section...
"Supported Interfaces" on page 20-3
"Supported Hardware" on page 20-4

Supported Interfaces

The Instrument Control Toolbox supports the use of instruments and communication via the following interfaces. The table lists the interface support by platform. Notes after the table contain more specific information.

Feature	64-bit MATLAB on Windows	32-bit MATLAB on Windows	64-bit MATLAB on Mac OS	64-bit MATLAB on Linux
Serial	supported	supported	supported	supported
TCP/IP	supported	supported	supported	supported
UDP	supported	supported	supported	supported
VISA [3]	supported [1]	supported [1]		supported [1]
GPIB [4]	supported [1]	supported [1]		supported [1]
I2C [5]	supported [1]	supported [1]	supported [1]	supported [1]
SPI [5]	supported [1]	supported [1]	supported [1]	supported [1]
Bluetooth [6]	supported	supported	supported [7]	
Quick-Control Oscilloscope and Quick-Control Function Generator	supported [2]	supported [2]	supported [2]	supported [2]
MATLAB Instrument Drivers	supported	supported	supported	supported
MATLAB Instrument Drivers made	supported [1]	supported [1]		

Feature	64-bit MATLAB on Windows	32-bit MATLAB on Windows	64-bit MATLAB on Mac OS	64-bit MATLAB on Linux
using IVI-C drivers and Instrument Wrappers for IVI-C drivers				
MATLAB Instrument Drivers made using IVI-COM drivers and Instrument Wrappers for IVI-COM drivers		supported [1]		

Table Notes

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

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

3. Requires Agilent, National Instruments, Tektronix, or TAMS VISA compliant with VISA specification 5.0 or higher.

4. Requires Advantech, Agilent, CEC, Contec, ICS Electronics, IOtech, Keithley, MCC, or National Instruments hardware and driver.

5. Requires Aardvark or National Instruments hardware and driver.

6. Bluetooth Serial Port Profile only.

7. Bluetooth on 64-bit MATLAB on Mac OS 10.7 and earlier versions only.

Supported Hardware

See http://www.mathworks.com/hardware-support/instrument-control-software.html for a complete list of supported hardware.

Troubleshooting SPI Interface

In this section...
"Supported Platforms" on page 20-5
"Adaptor Requirements" on page 20-6
"Configuration and Connection" on page 20-7

Serial Peripheral Interface (SPI) is a synchronous serial data link standard that operates in full duplex mode. It is commonly used in the test and measurement field. Common uses include communicating with micro controllers, EEPROMs, A2D devices, embedded controllers, etc.

Instrument Control Toolbox SPI support lets you open connections with individual chips and to read and write over the connections to individual chips using an Aardvark or NI-845x host adaptor. The primary uses for the `spi` interface involve the `write`, `read`, and `writeAndRead` functions for synchronously reading and writing binary data.

Supported Platforms

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface.

The SPI interface is supported on these platforms when used with the Aardvark host adaptor:

- Linux — Red Hat Enterprise Linux 4 and 5 with kernel 2.6, and possibly SUSE and Ubuntu distributions.
- Mac OS X 64-bit — Intel versions of Mac OS X 10.5 Leopard, 10.6 Snow Leopard, 10.7 Lion, and 10.8 Mountain Lion.
- Microsoft Windows 32-bit and 64-bit — Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), Windows 7 (32-bit and 64-bit), and Windows 8 (32-bit and 64-bit).

The SPI interface is supported on these platforms when used with the NI-845x host adaptor:

- Microsoft Windows 32-bit and 64-bit — Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), Windows 7 (32-bit and 64-bit), and Windows 8 (32-bit and 64-bit).

Adaptor Requirements

You need either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `spi` interface. The following sections describe requirements for each option.

Aardvark-specific Requirements

To use the SPI interface with the Aardvark adaptor, download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the Aardvark driver installed, see "Install the Total Phase Aardvark I2C/SPI Support Package" on page 14-9 to install it via the Support Package Installer.

Install the Aardvark Software API and Shared Library appropriate for your operating system.

The `aardvark.dll` file that comes with the Total Phase Aardvark adaptor board must be available in one of these locations for use on Windows platforms:

- Location where MATLAB was started from (bin folder)
- MATLAB current folder (PWD)
- Windows folder `C:\winnt` or `C:\windows`
- Folders listed in the path environment variable

For use on Linux platforms, the `aardvark.so` file that comes with the Total Phase Aardvark adaptor board must be in your MATLAB path.

NI-845x-specific Requirements

To use the SPI interface with the NI-845x adaptor, download the hardware support package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this support package.

If you do not have the NI-845x driver installed, see "Install the NI-845x I2C/SPI Interface Support Package" on page 14-8 to install it via the Support Package Installer.

Configuration and Connection

1 Make sure that you have the correct instrument driver installed for your device. Refer to your device's documentation and the vendor's web site.

2 Make sure your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 20-3.

3 You must have a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the SPI interface. Install the appropriate support package if you have not already. See "Adaptor Requirements" on page 20-6.

Make sure that your SPI adaptor board is plugged into the computer running MATLAB. You can verify that you have one of the adaptors installed by using the `instrhwinfo` function with the `spi` interface name.

```
>> instrhwinfo('spi')

ans =

  HardwareInfo with properties:

    SupportedVendors: {'aardvark'  'ni845x'}
    InstalledVendors: {'ni845x'}
```

If you do not see either `aardvark` or `ni845x` listed, you need to install one of the support packages or install the driver directly from the vendor.

4 Make sure that Instrument Control Toolbox recognizes your device, by using the `instrhwinfo` function with the `spi` interface name, and your adaptor name, either `aardvark` or `ni845x`. For example:

```
>> instrhwinfo('spi' , 'Aardvark')

ans =

            VendorName: 'aardvark'
     VendorDescription: 'Total Phase I2C/SPI Driver'
     VendorLibraryName: 'aardvark.dll'
     InstalledBoardIds: {[0]}
     BoardSerialNumbers: {'2237722838'}
     ObjectConstructors: {'spi('aardvark', 0, 0)'}
```

You will need the information displayed to create the spi object. If your device is not displayed, check the previous steps.

5 Make sure you can create the spi object. You must provide three arguments to create the object. BoardIndex and Port are both usually 0, and Vendor is either 'aardvark' or 'ni845x'. This example uses a SPI object called S that communicates to an EEPROM chip. Create the object using the BoardIndex and Port numbers, which are 0 in both cases.

```
% Vendor = aardvark
% BoardIndex = 0
% Port = 0

S = spi('aardvark', 0, 0);
```

6 If you do not get an error, the object was created successfully. To verify, you can look at the object properties, using the name you assigned to the object, S in this case.

```
>> disp(S)
SPI Object :

    Adapter Settings
        BoardIndex:             0
        BoardSerial:            2237722838
        VendorName:             aardvark

    Communication Settings
        BitRate:                1000000 Hz
        ChipSelect:             0
        ClockPhase:             FirstEdge
        ClockPolarity:          IdleLow
        Port:                   0

    Communication State
        ConnectionStatus:       Disconnected

    Read/Write State
        TransferStatus:         Idle
```

7 Make sure you can connect to the device, using the `connect` function with the object name.

```
connect(S);
```

If you do not get an error, the connection was made successfully. If you do get an error, follow the steps in the error message and/or check the previous steps listed here.

8 When you have connected, you can communicate with your device. See "Transmitting Data Over the SPI Interface" on page 10-9 for an example of reading and writing to a chip.

Troubleshooting I2C Interface

In this section...
"Supported Platforms" on page 20-10
"Adaptor Requirements" on page 20-11
"Configuration and Connection" on page 20-12

I2C, or Inter-Integrated Circuit, is a chip-to-chip interface supporting two-wire communication. Instrument Control Toolbox I2C support lets you open connections with individual chips and to read and write over the connections to individual chips.

The Instrument Control Toolbox I2C interface lets you do chip-to-chip communication using an Aardvark or NI-845x host adaptor. Some applications of this interface include communication with SPD EEPROM and NVRAM chips, communication with SMBus devices, controlling accelerometers, accessing low-speed DACs and ADCs, changing settings on color monitors using the display data channel, changing sound volume in intelligent speakers, reading hardware monitors and diagnostic sensors, visualizing data sent from an I2C sensor, and turning on or off the power supply of system components.

Supported Platforms

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the I2C interface.

The I2C interface is supported on these platforms when used with the Aardvark host adaptor:

- Linux – The software works with Red Hat Enterprise Linux 4 and 5 with kernel 2.6. It may also be successful with SuSE and Ubuntu distributions.

- Mac OS X 64-bit – The software is supported on Intel versions of Mac OS X 10.5 Leopard and 10.6 Snow Leopard.

- Microsoft Windows 32-bit and 64-bit — Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), Windows 7 (32-bit and 64-bit), and Windows 8 (32-bit and 64-bit).

The I2C interface is supported on these platforms when used with the NI-845x host adaptor:

- Microsoft Windows 32-bit and 64-bit — Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), Windows 7 (32-bit and 64-bit), and Windows 8 (32-bit and 64-bit).

For updates to the list of currently supported platforms for MATLAB, see http://www.mathworks.com/support/sysreq/current_release/.

Adaptor Requirements

You need to have either a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the I2C interface. The following sections describe requirements for each option.

Aardvark-specific Requirements

To use the I2C interface with the Aardvark adaptor, download the Hardware Support Package to obtain the latest driver, if you do not already have the driver installed. If you already have the latest driver installed, you do not need to download this Support Package.

If you do not have the Aardvark driver installed, see "Install the Total Phase Aardvark I2C/SPI Support Package" on page 14-9 to install it via the Support Package Installer.

Install the Aardvark Software API and Shared Library appropriate for your operating system.

The `aardvark.dll` file that comes with the Total Phase Aardvark adaptor board must be available in one of these locations for use on Windows platforms:

- Location where MATLAB was started from (bin folder)
- MATLAB current folder (PWD)
- Windows folder `C:\winnt` or `C:\windows`
- Folders listed in the path environment variable

For use on Linux platforms, the `aardvark.so` file that comes with the Total Phase Aardvark adaptor board must be in your MATLAB path.

NI-845x-specific Requirements

To use the I2C interface with the NI-845x adaptor, download the hardware support package to obtain the latest driver, if you do not already have the driver installed. If

you already have the latest driver installed, you do not need to download this support package.

If you do not have the NI-845x driver installed, see "Install the NI-845x I2C/SPI Interface Support Package" on page 14-8 to install it via the Support Package Installer.

Configuration and Connection

1 Make sure that you have the correct instrument driver installed for your device. Refer to your device's documentation and the vendor's web site.

2 Make sure your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 20-3.

3 You must have a Total Phase Aardvark host adaptor or an NI-845x adaptor board installed to use the `i2c` interface. Install the appropriate support package if you have not already. See "Adaptor Requirements" on page 20-11.

Make sure that your I2C adaptor board is plugged into the computer running MATLAB. You can verify that you have one of the adaptors installed by using the `instrhwinfo` function with the `i2c` interface name.

```
>> instrhwinfo('i2c')

ans =

    InstalledAdaptors: 'Aardvark'
        JarFileVersion: 'Version 3.0.0'
```

If you do not see either `aardvark` or `ni845x` listed, you need to install one of the support packages or install the driver directly from the vendor.

4 Make sure that Instrument Control Toolbox recognizes your device, by using the `instrhwinfo` function with the `i2c` interface name, and your adaptor name, either `aardvark` or `ni845x`. For example:

```
instrhwinfo('i2c', 'Aardvark')

ans =

        AdaptorDllName: [1x127 char]
```

```
        AdaptorDllVersion: 'Version 3.0.0'
              AdaptorName: 'aardvark'
         InstalledBoardIds: 0
    ObjectConstructorName: 'i2c('aardvark', BoardIndex, RemoteAddress);'
            VendorDllName: 'aardvark.dll'
  VendorDriverDescription: 'Total Phase I2C Driver'
```

You will need the information displayed to create the i2c object. If your device is not displayed, check the previous steps.

5 Make sure you can create the i2c object. You must provide three arguments to create the object. BoardIndex is usually 0, and Vendor is either aardvark or ni845x. The RemoteAddress is specific to your device. Read the documentation of the chip in order to know what the remote address is. For example, in this case we create an object to communicate with an eeprom chip at remote address 50h:

```
% Vendor = aardvark
% BoardIndex = 0
% RemoteAddress = 50

eeprom = i2c('aardvark', 0, '50h');
```

Tip You can also see what the remote address of the chip is by scanning for instruments in the Test & Measurement Tool. In the tool, right-click the **I2C** node and select **Scan for I2C adaptors**. Any chips found by the scan are listed in the hardware tree. The listing includes the remote address of the chip.

6 If you do not get an error, the object was created successfully. To verify, you can look at the object properties, using the name you assigned to the object, eeprom in this case.

```
>> disp(eeprom)

    I2C Object : I2C-0-50h

    Communication Settings
        BoardIndex          0
        BoardSerial         2237482577
        BitRate:            100
        RemoteAddress:      50h
        Vendor:             aardvark


    Communication State
        Status:             open
        RecordStatus:       off

    Read/Write State
        TransferStatus:     idle
        BytesAvailable:     0
        ValuesReceived:     16
        ValuesSent:         15
```

7 Make sure you can connect to the device, using the `fopen` function with the object name.

```
fopen(eeprom);
```

If you do not get an error, the connection was made successfully. If you do get an error, follow the steps in the error message and/or check the previous steps listed here.

8 When you have connected, you can communicate with your device. See "Transmitting Data Over the I2C Interface" on page 9-9 for an example of reading and writing to a chip.

Troubleshooting Bluetooth Interface

In this section...
"Supported Platforms" on page 20-15
"Adaptor Requirements" on page 20-15
"Configuration and Connection" on page 20-17
"Other Troubleshooting Tips for Bluetooth" on page 20-20
"Troubleshoot Bluetooth Interface in Test & Measurement Tool" on page 20-21

The Instrument Control Toolbox Bluetooth interface lets you connect to devices over the Bluetooth interface and to transmit and receive ASCII and binary data. Instrument Control Toolbox supports the Bluetooth Serial Port Profile (SPP). You can identify any SPP Bluetooth device and establish a two-way connection with that device.

Bluetooth is an open wireless technology standard for exchanging data over short distances using short wavelength radio transmissions from fixed and mobile devices over a packet-based protocol. Bluetooth provides a secure way to connect and exchange information between devices such as Lego Mindstorm NXT robots, USB Bluetooth adaptors (dongles), wireless sensors, mobile phones, faxes, laptops, computers, printers, GPS receivers, etc.

Supported Platforms

The Bluetooth interface is supported on these platforms:

- Mac OS 10.7 and earlier versions only 64-bit
- Microsoft Windows 32-bit
- Microsoft Windows 64-bit

Adaptor Requirements

Instrument Control Toolbox supports the Bluetooth Serial Port Profile (SPP). You can identify any SPP Bluetooth device and establish a two-way connection with that device.

Instrument Control Toolbox can communicate with Bluetooth devices via an adaptor. In this example, a USB Bluetooth adaptor is plugged into the computer. It can identify

Bluetooth devices within range when queried. In order to communicate with instruments, you need to perform a pairing in the adaptor software. Note that some devices, such as many laptop computers, do not need to use an adaptor since they have one built in.

The following shows the software interface of an adaptor where two of the devices in range have been paired – a smart phone with Bluetooth enabled, and a Lego Mindstorm NXT robot. As you can see, the "friendly name" or display name of the smart phone is simply `iPhone` and the name of the NXT robot is `C3PO`. In Instrument Control Toolbox this friendly name is the Bluetooth `RemoteName` property.

On Windows 7 64-bit platforms, you can use only one Bluetooth adaptor at a time. If you connect another adaptor, it will fail with a "Device Driver Installation Failed" error.

Some adaptors support multiple devices:

- The Bluetooth adaptor that comes with the LEGO Mindstorm kit (Abe – Model: UB22S) supports connection to only one Bluetooth device at a time.
- IO Gear – Models GBU421 and GBU311 support communication with multiple Bluetooth devices.
- Targus – Model ACB10US supports communication with multiple Bluetooth devices.
- Motorola – Model SYN1244B supports communication with multiple Bluetooth devices.

- D-Link – Model DBT-120 supports communication with multiple Bluetooth devices.

If a Bluetooth adaptor is removed and a different one plugged in, all Bluetooth devices have to be paired again with your PC. If the same adaptor is removed and plugged back in, then you do not need to pair the devices again. If another adaptor of the same vendor is plugged in, then the devices which had been cached when that adaptor was used are seen in the cache.

Configuration and Connection

1 Make sure that you have the correct instrument driver installed for your device. Refer to your device's documentation and the vendor's web site.

2 Make sure your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 20-3.

3 Make sure that the Bluetooth service on the device is turned on.

4 Instrument Control Toolbox can communicate with Bluetooth devices via an adaptor, either a USB Bluetooth adaptor that is plugged into the computer, or a built in adaptor in the device. See "Adaptor Requirements" on page 20-15.

   Make sure you have performed the pairing between the device and the computer running MATLAB by using either a USB adaptor or a built-in adaptor.

5 You can verify that MATLAB can see your device by using the `instrhwinfo` function with the `Bluetooth` interface name.

```
>> instrhwinfo('Bluetooth')

ans =

        RemoteNames: {5x1 cell}
          RemoteIDs: {5x1 cell}
    BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
     JarFileVersion: 'Version 3.0.0'

>> b = instrhwinfo('Bluetooth')

b =

        RemoteNames: {5x1 cell}
          RemoteIDs: {5x1 cell}
    BluecoveVersion: 'BlueCove-2.1.1-SNAPSHOT'
     JarFileVersion: 'Version 3.0.0'
```

In this example, `instrhwinfo` returned a cell array of five Bluetooth devices that are in the range of the adaptor on the computer running Instrument Control Toolbox. Then indexing into the `RemoteNames` property shows the five devices. You can see that `iPhone` and `C3PO` appear in the list.

```
>> b.RemoteNames

ans =

    'iPhone'
    'mprocopi-maci'
    'C3PO'
    ''
    'Emulator'
```

If a device shows an empty string for `RemoteName`, then that device does not have a friendly name associated with it. To communicate with that device, you need to use the `RemoteID` property.

```
>> b.RemoteIDs

ans =

    'E0F847D773E4'
    '001F5BDC815E'
    '0016530FD63D'
    '0021BA74F3DD'
    '00000000000'
```

The `RemoteIDs` display in the same order as the `RemoteNames`, so in this case, the fourth ID in the list, `'0021BA74F3DD'`, could be used for the device that shows no `RemoteName`. You can use either `RemoteName` or `RemoteID` to communicate with a device.

You can verify that MATLAB can see your device by using the `instrhwinfo` function with the `RemoteName` property. In this example, `C3PO` is the remote name of the NXT robot and is shown in the output.

```
instrhwinfo('Bluetooth','C3PO')
```

The Instrument Control Toolbox displays the device information.

```
>> instrhwinfo('Bluetooth','C3PO')

ans =

              RemoteName: 'C3PO'
                DeviceID: '0016530FD63D'
     ObjectConstructorName: {'Bluetooth('C3PO', 1);'}
                Channels: {'1'}
```

6 Make sure you can create the `Bluetooth` object. You must provide two arguments to create the object, either the `RemoteName` or the `RemoteID`, and a channel number. You can get the channel number from your device's documentation or from looking at the **Bluetooth** node in the Test & Measurement Tool. For example, create a Bluetooth object called `bt` using channel 1 of the NXT device.

```
bt = Bluetooth('C3PO', 1);
```

If you do not get an error, the object was created successfully.

7 Make sure you can connect to the device, using the `fopen` function with the object name.

```
fopen(bt)
```

If you do not get an error, the connection was made successfully. If you do get an error, follow the steps in the error message and/or check the previous steps listed here.

8 When you have connected, you can communicate with your device. See in "Transmitting Data Over the Bluetooth Interface" on page 8-10 for an example of communicating with a device. See "Other Troubleshooting Tips for Bluetooth" on page 20-20 for tips about communication issues after initial connection.

Other Troubleshooting Tips for Bluetooth

These tips may be relevant to your use of this feature.

Remove or change adaptors

If a Bluetooth adaptor is removed and a different one plugged in, all Bluetooth devices have to be paired again with your PC. If the same adaptor is removed and plugged back in, then you do not need to pair the devices again. If another adaptor of the same vendor is plugged in, then the devices which had been cached when that adaptor was used are seen in the cache.

Use of instrhwinfo

If a Bluetooth device is already cached, but it is OFF when MATLAB is started, and if `instrhwinfo` is called on this device, then `ObjectConstructorName` and `Channel` are returned as a null string.

If a Bluetooth device is already cached and is ON when MATLAB is started, and it is later switched OFF, if `instrhwinfo` is called on this device, then `ObjectConstructorName` and `Channel` return the correct values.

Device out of range

If you create a Bluetooth object for any Bluetooth device and the connection is open, and then the device goes out of range, the status of the object would still be `open`. When the device comes into range again, you need to `fclose` the object and `fopen` it again for communication to continue.

Change batteries on device

If you create a Bluetooth object, for a Lego Mindstorm NXT robot for example, and the connection is open, and then the batteries of robot run out, then the status of the object would still be `open`. If you then replace the batteries, you need to `fclose` the object and `fopen` it again for communication to continue.

Restart device after scan in Test & Measurement Tool

When using the Bluetooth support in the Test & Measurement Tool, please note that you may need to restart your device after you have done the scan. For any Lego Mindstorm robot to be identified correctly, it has to be restarted after scanning. You may also have to restart other Bluetooth devices after the scan as well.

Troubleshoot Bluetooth Interface in Test & Measurement Tool

To use the Bluetooth support in the Test & Measurement Tool, select the **Bluetooth** node in the instrument tree and right-click **Scan for bluetooth devices**.

Note: When using the Bluetooth support in the Test & Measurement Tool, please note that you may need to restart your device after you have done the scan. For any Lego Mindstorm robot to be identified correctly, it has to be restarted after scanning. You may also have to restart other Bluetooth devices after the scan as well.

If you are having trouble using the Bluetooth interface in the Test & Measurement Tool, try these steps.

- Check that the Bluetooth device supports the Serial Port Profile (SPP). We do not support other Bluetooth profiles such as File Transfer Profile (FTP).
- Make sure that the Bluetooth service on the device is turned on.
- Make sure that the Bluetooth device is paired with your computer.
- If you are using a Lego Mindstorm NXT brick, note that the NXT brick has to be restarted after scanning for Bluetooth devices from the Test & Measurement Tool.
- If you still cannot connect to the Bluetooth device, try unplugging and replugging the Bluetooth adaptor.

Troubleshooting Serial Port Interface

In this section...
"Supported Platforms" on page 20-23
"Adaptor Requirements" on page 20-24
"Configuration and Connection" on page 20-24
"Other Troubleshooting Tips for Serial Port" on page 20-26

Serial communication is a low-level protocol for communicating between two or more devices. Normally, one device is a computer, and the other device can be another computer or a:

- modem
- printer
- scientific instrument such as an oscilloscope or a function generator

The serial port sends and receives bytes of information in a serial fashion — 1 bit at a time. These bytes are transmitted using either a binary format or a text (ASCII) format.

For many serial port applications, you can communicate with your instrument without detailed knowledge of how the serial port works. Communication is established through a serial port object, which you create in the MATLAB workspace.

Supported Platforms

The serial port interface is supported on these platforms:

- Linux 64-bit
- Mac OS X 64-bit
- Microsoft Windows 32-bit
- Microsoft Windows 64-bit

The serial port interface is supported on the same platforms as MATLAB. For updates to the list of currently supported platforms, see http://www.mathworks.com/support/sysreq/current_release/.

Adaptor Requirements

Use RS-232 interface standard with the serial port communication. Over the years, several serial port interface standards for connecting computers to peripheral devices have been developed. These standards include RS-232, RS-422, and RS-485 — all of which are supported by the serial port object. Of these, the most widely used standard is RS-232, which stands for Recommended Standard number 232.

In this guide, it is assumed you are using the RS-232 standard.

You need to connect the two devices with a serial cable. For more information, see "Connecting Two Devices with a Serial Cable" on page 6-3.

Serial ports consist of two signal types: *data signals* and *control signals*. To support these signal types, as well as the signal ground, the RS-232 standard defines a 25-pin connection. However, most PCs and UNIX platforms use a 9-pin connection. In fact, only three pins are required for serial port communications: one for receiving data, one for transmitting data, and one for the signal ground. For more information, see "Serial Port Signals and Pin Assignments" on page 6-4.

Configuration and Connection

1 Make sure that you have the correct instrument driver installed for your device. Refer to your device documentation and the vendor website.

2 Make sure that your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 20-3.

3 Make sure that Instrument Control Toolbox recognizes your serial ports, by using the `instrhwinfo` function with the `serial` interface name. For example:

```
instrhwinfo('serial')

ans =

 HardwareInfo with properties:

     AvailableSerialPorts: {'COM1'}
           JarFileVersion: 'Version 3.7'
    ObjectConstructorName: {'serial('COM1');'}
               SerialPorts: {'COM1'}
```

If your computer has more than one serial port, your output would look like this:

```
info = instrhwinfo('serial')

info =

 HardwareInfo with properties:

     AvailableSerialPorts: {2x1 cell}
           JarFileVersion: 'Version 3.7'
    ObjectConstructorName: {2x1 cell}
               SerialPorts: {2x1 cell}
```

4   In this example the output listed two ports. List the available serial ports:

```
info.AvailableSerialPorts

ans =

    'COM1'
    'COM2'
```

Tip You can also use Windows device manager to see a list of available serial ports.

5   Make sure you can create your serial port object. You must provide one argument to create the object, the name of an available port. For example, create a serial object called s using port COM1.

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

If you do not get an error, the object was created successfully.

6   Make sure you can connect to the device, using the fopen function with the object name.

```
fopen(s);
```

If you do not get an error, the connection was made successfully. If you do get an error, follow the steps in the error message and/or check the previous steps listed here.

7   When you have connected, you can communicate with your device. If you have problems sending or receiving, you may need to configure communication settings such as BaudRate, DataBits, Parity, StopBits, or Terminator. Make sure you configure these communication parameters to match those of the connected device. For more information on using these properties, see .

See "Writing and Reading Text Data" on page 6-25 and "Writing and Reading Binary Data" on page 6-29 for communication examples.

Other Troubleshooting Tips for Serial Port

Verify Port

Verify that the serial (COM) port is listed in Windows Control Panel > Device Manager > Ports.

Sending and Receiving

If you have problems sending or receiving, you may need to configure communication settings such as `BaudRate`, `DataBits`, `Parity`, `StopBits`, or `Terminator`. Make sure you configure these communication parameters to match those of the connected device. For more information on using these properties, see .

Buffer Settings

If you have problems sending or receiving, you may need to configure buffer settings.

The `ReadAsyncMode` property should always be set to `continuous`. Using `manual` usually results in data loss.

`InputBufferSize` needs to be set to larger than the largest chunk of data the toolbox will receive. If it is too small, data will be dropped. For continuous streams, try using a bigger buffer size. The buffer should be large enough to hold at least one second of data. The buffer should also be at least twice the read size (the number of bytes read by `fread/fscanf`).

VISA

For serial communication, you can also use VISA with a VISA resource name, as defined in a VISA vendor utility, such as Agilent Connection Expert.

Third-party Software

For troubleshooting serial port communication, you can also use a third-party serial communication software, such as PuTTY or Tera Term, to isolate the issue.

Incorrect Data

When doing binary data communication with `fread` and `fwrite`, make sure the correct data type – for example `int16`, `uint16`, `double` – is being used with `fread` and `fwrite`. You should use the same data type as the instrument uses.

If reading and writing data types other than `uint8` or `int8`, make sure the `ByteOrder` is correct.

20-27

Troubleshooting GPIB Interface

GPIB is a standardized interface that allows you to connect and control multiple devices from various vendors. GPIB is also referred to by its original name HP-IB, or by its IEEE designation IEEE-488.

Some of the GPIB functionality is required for all GPIB devices, while other GPIB functionality is optional. Refer to your device documentation for a complete list of its GPIB capabilities and its command set.

Supported Platforms

The GPIB interface support is dependent on support by third-party vendor driver for the hardware on the supported platforms.

The GPIB interface is supported on these platforms:

- Linux – The software works with Red Hat Enterprise Linux 4 and 5 with kernel 2.6. It may also be successful with SuSE and Ubuntu distributions.
- Microsoft Windows 32-bit and 64-bit — Windows XP (SP2 or later, 32-bit and 64-bit), Windows Vista (32-bit and 64-bit), Windows 7 (32-bit and 64-bit), and Windows 8 (32-bit and 64-bit).

Adaptor Requirements

Instrument hardware and driver

The GPIB interface support requires Advantech, Agilent, CEC, Contec, ICS Electronics, IOtech, Keithley, MCC, or National Instruments adaptor board and driver.

Bus and connector

You need a bus and connector to communicate with GPIB instruments. The GPIB bus is a cable with two 24-pin connectors that allow you to connect multiple devices to each other. For more information, see "Bus and Connector" on page 4-3.

GPIB devices

Each GPIB device must be some combination of a Talker, a Listener, or a Controller. A Controller is typically a board that you install in your computer. Talkers and Listeners are typically instruments such as oscilloscopes, function generators, multimeters, and so on. Most modern instruments are both Talkers and Listeners. Each Controller is identified by a unique board index number. Each Talker/Listener is identified by a unique primary address ranging from 0 to 30, and by an optional secondary address, which can be 0 or can range from 96 to 126. for more information, see "GPIB Devices" on page 4-3.

GPIB data

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

For information on the types of lines and GPIB pin and signal assignments, see "GPIB Lines" on page 4-4.

Configuration and Connection

1   Make sure that you have the correct instrument driver installed for your device. Refer to your device's documentation and the vendor's web site.

2   Make sure your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 20-3.

3   You must have a GPIB board and instrument installed to use the GPIB interface.

Make sure that your GPIB adaptor board is plugged into the computer running MATLAB. You can verify that you have an adaptor board installed by using the `instrhwinfo` function with the `gpib` interface name. Allowed adaptors include `advantech`, `agilent`, `cec`, `isc`, `iotech`, `keithley`, `mcc`, and `ni`.

```
instrhwinfo('gpib')

ans =
    InstalledAdaptors: {'advantech', 'agilent'}
       JarFileVersion: 'Version 14.3.0'
```

This example shows two installed vendors. The vendor name is for the board manufacturer, not your instrument manufacturer. If you do not see any installed vendors, you must install a board and its driver.

4   Make sure that Instrument Control Toolbox recognizes your device, by using the `instrhwinfo` function with the `gpib` interface name, and your adaptor name. For example:

```
instrhwinfo('gpib', 'agilent')

ans =
             AdaptorDllName: [1x91 char]
          AdaptorDllVersion: 'Version 14.3.0'
                AdaptorName: 'agilent'
           InstalledBoardIds: [8 32]
       ObjectConstructorName: {'gpib('agilent', 8, 6);'}
              VendorDllName: 'sicl32.dll'
     VendorDriverDescription: 'Agilent Technologies GPIB Driver'
```

`InstalledBoardIds` should list all installed boards. This is not the GPIB address of the board. If your board is not listed, try to use the board manufacturer's software to use the board.

`ObjectConstructorName` should have one entry for every connected instrument. If your instrument is not listed, make sure the instrument is on and configured for GPIB communication.

5   Make sure you can create the `gpib` object. Each GPIB object is associated with one controller and one instrument. You must provide three arguments to create the object. `Vendor` is the adaptor board vendor. `BoardIndex` and `PrimaryAddress` are the second and third arguments. For example, to create a GPIB object associated with a National Instruments controller with board index `0`, and an instrument with primary address `1`:

```
% Vendor = ni
% BoardIndex = 0
% PrimaryAddress = 1

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.

6   If you do not get an error, the object was created successfully. To verify, you can look at the object properties, using the name you assigned to the object, `g` in this case.

```
disp(g)

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
```

7 Make sure you can connect to the device, using the `fopen` function with the object name.

```
fopen(g);
```

If you do not get an error, the connection was made successfully. If you do get an error, follow the steps in the error message and/or check the previous steps listed here.

8 When you have connected, you can communicate with your device. See "Writing and Reading Data" on page 4-18 for an example of reading and writing to a chip.

Other Troubleshooting Tips for GPIB

Verify in Vendor Utility

Verify correct operation in a vendor utility, such as Agilent Connection Expert or National Instruments Measurement and Automation Explorer (MAX).

Both Agilent and NI installed

If you have both Agilent and NI GPIB drivers installed, you may get conflicts. Try disabling the other vendor drivers and trying the vendor you want to use again.

EOIMode, EOSCharCode, and EOSMode

`EOIMode` is the default and should be left `on` most of the time. However, some instruments might require `EOIMode` to be turned `off`. Use the correct parameter, such as `EOSCharCode` or `EOSMode` when necessary.

VISA

You can almost always use the `visa` interface with a VISA GPIB resource name instead of the `gpib` interface if you run into an issue. The only limitation of using VISA is that you cannot use the `spool` function.

Incorrect command sent

A common problem in using the GPIB interface is that an incorrect command is sent to the instrument. Try:

```
query(g,'*IDN?')
```

to verify communication with the instrument.

Most newer instruments recognize this command, since it is part of the GPIB standard. You should also look at the instrument's manual to see which commands it recognizes.

Byte Order

You can configure `ByteOrder` to be `littleEndian` or `bigEndian`. For GPIB, `ByteOrder` refers to the order in which the bytes in a multi-byte data type values are transmitted on the communication bus. You can use the `swapbytes` command to troubleshoot issues with `ByteOrder`. 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.

Incorrect Data

Make sure the correct data type – for example `int16`, `uint16`, `double` – is being used with `fread` and `fwrite`. You should use the same data type as the instrument uses.

If reading and writing data types other than `uint8` or `int8`, make sure the `ByteOrder` is correct.

Troubleshooting TCP/IP Interface

In this section...
"Supported Platforms" on page 20-34
"Configuration and Connection" on page 20-34
"Other Troubleshooting Tips for TCP/IP" on page 20-35

Transmission Control Protocol (TCP) is a transport protocol layered on top of the Internet Protocol (IP) and is one of the most highly used networking protocols. With Instrument Control Toolbox you can use raw socket communication and connect to remote hosts to read and write data. For example, you could use it to acquire data from a remote weather station, and plot the data.

Supported Platforms

TCP/IP is supported on these platforms:

- Linux 64-bit
- Mac OS X 64-bit
- Microsoft Windows 32-bit
- Microsoft Windows 64-bit

The TCP/IP interface is supported on the same platforms as MATLAB. For updates to the list of currently supported platforms, see http://www.mathworks.com/support/sysreq/current_release/.

Configuration and Connection

1   Make sure that you have the correct instrument driver installed for your device. Refer to your device documentation and the vendor website.
2   Make sure that your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 20-3.
3   Make sure that Instrument Control Toolbox recognizes your device, by using the `instrhwinfo` function with the `tcpip` interface name. For example:

```
instrhwinfo('tcpip')

ans =
```

```
HardwareInfo with properties:

          LocalHost: {6x1 cell}
     JarFileVersion: 'Version 3.7'
```

4   Make sure you can create your TCP/IP object. You create a TCP/IP object with the `tcpip` function, which requires the name of the remote host as an input argument. In most cases, you also need to specify the remote port value. If you do not specify the remote port, then 80 is used.

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, you use the remote host name of the instrument and the port number, which can be found in the instrument's documentation.

```
t = tcpip('sonytekawg.yourdomain.com',4000);
```

5   Make sure you can connect to the device, using the `fopen` function with the object name.

```
fopen(t);
```

If you do not get an error, the connection was made successfully. The `fopen` function errors if the socket connection cannot be made to the TCP/IP server. If `fopen` works, the computer and port number are correct and a TCP/IP server is running on the other end. If you do get an error, follow the steps in the error message and/or check the previous steps listed here.

6   When you have connected, you can communicate with your device. If sending and receiving does not work, you can check the following:

- Make sure the data is being sent in the format expected by the server.
- If you connect to a web server, you may need to send HTTP `get` or `post` commands. You can also use the `urlread` or `webread` functions to communicate with web servers.
- Many TCP/IP servers expect header information inside the TCP/IP packet.

See "Communicating with a Remote Host" on page 7-6 for an example of communication over TCP/IP.

Other Troubleshooting Tips for TCP/IP

VISA

You can also use the `visa` interface with a VISA TCP/IP resource name instead of the `tcpip` interface for TCP/IP communication with instruments.

Incorrect Data

Make sure the correct data type – for example `int16`, `uint16`, `double` – is being used with `fread` and `fwrite`. You should use the same data type as the instrument uses.

If reading and writing data types other than `uint8` or `int8`, make sure the `ByteOrder` is correct.

Troubleshooting UDP Interface

In this section...
"Supported Platforms" on page 20-37
"Configuration and Connection" on page 20-37

User Datagram Protocol (UDP or UDP/IP) is a transport protocol layered on top of the Internet Protocol (IP). UDP is a connectionless protocol. An application using UDP prepares a packet and sends it to the receiver's address without first checking to see if the receiver is ready to receive a packet. If the receiving end is not ready to receive a packet, the packet is lost.

Supported Platforms

UDP is supported on these platforms:

- Linux 64-bit
- Mac OS X 64-bit
- Microsoft Windows 32-bit
- Microsoft Windows 64-bit

The UDP interface is supported on the same platforms as MATLAB. For updates to the list of currently supported platforms, see http://www.mathworks.com/support/sysreq/current_release/.

Configuration and Connection

1 Make sure that you have the correct instrument driver installed for your device. Refer to your device documentation and the vendor website.

2 Make sure that your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 20-3.

3 Make sure that Instrument Control Toolbox recognizes your device, by using the instrhwinfo function with the udp interface name. For example:

```
instrhwinfo('udp')

ans =
```

```
HardwareInfo with properties:

        LocalHost: {6x1 cell}
   JarFileVersion: 'Version 3.7'
```

4 Make sure you can create your UDP object. You create a UDP object with the `udp`
function, which 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.

Although UDP is a stateless connection, opening a UDP object with an invalid
host name will generate an error. You can configure property values during object
creation, such as the `LocalPort` property if you will use the object to read data from
the instrument. 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);
```

You can verify the object by looking at the object properties, using the name you
assigned to the object, `u` in this case.

```
disp(u)

UDP Object : UDP-127.0.0.1

Communication Settings
   RemotePort:        4012
   RemoteHost:        127.0.0.1
   Terminator:        'LF'

Communication State
   Status:            closed
   RecordStatus:      off

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

5 Make sure you can connect to the remote host, using the `fopen` function with the
object name.

```
fopen(u);
```

If the computer or host does not exist, you will get a warning. You can try to ping the computer to see if it is responding:

`!ping hostname`

6 When you have connected, you can communicate with your device. If sending and receiving does not work, you can check the following:

- Make sure the hostname/IP exists. Try `!ping hostname`.

- Make sure the `RemotePort` and `LocalPort` are correct. `RemotePort` is the port that the other computer or application is listening to for UDP traffic. `LocalPort` is the port on the computer that MATLAB is listening to for UDP traffic.

- Look at the setting of the `DatagramTerminateMode` property. The default setting of `on` makes UDP behave differently than other interfaces. Try both settings. You also may need to set other properties, such as buffer size. See "Functions and Properties" on page 7-31 (ASCII) and "Functions and Properties" on page 7-36 (Binary) for more information on properties and values.

- UDP is not a reliable protocol and packets can be dropped. You may need to try sending or receiving multiple times.

See "Reading and Writing ASCII Data over UDP" on page 7-31 and "Reading and Writing Binary Data over UDP" on page 7-36 for examples of communication over UDP and information on using properties.

Troubleshooting IVI, VISA, and the Quick-Control Interfaces

Virtual Instrument Software Architecture (VISA) is an industry standard defined by the IVI foundation for communicating with instruments regardless of the interface.

The IVI standard defines an open driver architecture, a set of instrument classes, and shared software components. With IVI you can use instruments interchangeability into multiple systems using standardized code.

You can use the Quick-Control Oscilloscope for any oscilloscope that uses an underlying IVI-C driver. You can use the Quick-Control Function Generator for any function generator that uses an underlying IVI-C driver.

In this section...
"Supported Platforms" on page 20-40
"Adaptor Requirements" on page 20-41
"Configuration and Connection" on page 20-44
"VISA Supported Vendor and Resource Names" on page 20-46

Supported Platforms

VISA is supported on these platforms:

- Linux 64-bit
- Mac OS X 64-bit
- Microsoft Windows 32-bit
- Microsoft Windows 64-bit

IVI is supported on these platforms:

- Microsoft Windows 32-bit
- Microsoft Windows 64-bit

The Quick Control interfaces are supported on these platforms:

- Microsoft Windows 32-bit
- Microsoft Windows 64-bit

Adaptor Requirements

IVI-C and IVI-COM

Instrument Control Toolbox software supports IVI-C and IVI-COM drivers, with class-compliant and instrument-specific functionality.

IVI class-compliant drivers support common functionality across a family of related instruments. Use class-compliant drivers to access the basic functionality of an instrument, and the ability to swap instruments without changing the code in your application. With an IVI instrument-specific driver or interface, you can access the unique functionality of the instrument. The instrument-specific driver generally does not accommodate instrument substitution.

For IVI-C drivers, you can use IVI-C class drivers and IVI-C specific drivers. Device objects you construct to call IVI-C class drivers offer interchangeability between similar instruments, and work with all instruments consistent with that class driver. Device objects you construct to call IVI-C specific drivers directly generally offer less interchangeability, but provide access to the unique methods and properties of a specific instrument.

Other things to note:

- IVI-COM is supported only on 32-bit MATLAB.
- Using an IVI driver with `icdevice` requires generating a MATLAB instrument driver (MDD) with `makemid` or using a prebuilt MDD driver.
- The IVI Foundation maintains a registry of drivers sortable by instrument model and driver type. See http://www.ivifoundation.org/registered_drivers/driver_registry.aspx.

Before you use IVI drivers in MATLAB, install:

- VISA
- IVI Shared components
- Required IVI drivers

IVI-C Wrappers

The IVI-C wrappers provide an interface to MATLAB for instruments running on IVI-C class-compliant drivers.

To use the wrapper you must have the following software installed.

- Windows 32-bit or Windows 64-bit

- VISA shared components
- VISA
- National Instruments compliance package NICP 4.1 or higher
- Your instrument driver

Quick Control Oscilloscope

You can use the Quick-Control Oscilloscope for any oscilloscope that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. You can also use it for Tektronix oscilloscopes.

To use the Quick-Control Oscilloscope for an IVI-C scope, you must have the following software installed.

- Windows 32-bit or Windows 64-bit platforms
- VISA shared components (installed by VISA)
- VISA
- National Instruments IVI compliance package NICP 4.1 or higher
- Your instrument's device-specific driver

By default, the driver used is Tektronix (`'tektronix'`). If your instrument is not supported by the default driver, specify a particular IVI-C Scope driver using the `driver` property on the `oscilloscope` object.

Note: As of release R2015a, most of these components are installed for you when you install the National Instruments VISA and ICP Interfaces Support Package. See "Install the National Instruments VISA and ICP Interfaces Support Package" on page 14-11.

Quick Control Function Generator

You can use the Quick-Control Function Generator for any function generator that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver.

To use the Quick-Control Function Generator for an IVI-C fgen, ensure the following software is installed:

- Windows 32-bit or Windows 64-bit platforms

- VISA shared components (installed by VISA)
- VISA
- National Instruments IVI compliance package NICP 4.1 or higher
- Your instrument's device-specific driver

By default, the driver used is `'Agilent332x0_SCPI'`. If your instrument is not supported by the default driver, specify a particular IVI-C Function Generator driver using the `driver` property on the `fgen` object.

Note: As of release R2015a, most of these components are installed for you when you install the National Instruments VISA and ICP Interfaces Support Package. See "Install the National Instruments VISA and ICP Interfaces Support Package" on page 14-11.

Configuration and Connection

1   Make sure that you have the correct instrument driver installed for your device. Refer to your device documentation and the vendor website.

2   Make sure that your device is supported in Instrument Control Toolbox. See "Is My Hardware Supported?" on page 20-3.

3   Make sure that Instrument Control Toolbox recognizes your device, by using the `instrhwinfo` function with the `visa` interface name. For example:

```
instrhwinfo('visa')

ans =
    InstalledAdaptors: {'agilent'}
        JarFileVersion: 'Version 2.8.0'
```

The VISA library should be listed. If it is not, make sure it is installed and it is a supported version.

You can also make sure your instrument is listed, by using `instrhwinfo` with the adaptor and vendor name, then looking at the `ObjectConstructorName` field.

```
visainfo = instrhwinfo('visa', 'agilent')

visainfo.ObjectConstructorName

ans =
```

```
'visa('agilent', 'ASRL1::INSTR');'
'visa('agilent', 'TCPIP0::172.31.57.119::inst0::INSTR');'
'visa('agilent', 'TCPIP0::hostname.net.com::inst0::INSTR');'
```

If the instrument is not listed, it is not configured properly in your vendor's VISA configuration utility. Each vendor has its own utility. See the table in "VISA Supported Vendor and Resource Names" at the end of this topic for a list of the vendor utilities.

4 You can also use `instrhwinfo` to find information on installed IVI drivers and shared components. For example:

```
instrhwinfo ('ivi')

ans =
                  LogicalNames: {'MainScope', 'FuncGen'}
                    ProgramIDs: {'TekScope.TekScope','Agilent33250'}
                       Modules: {'ag3325b', 'hpe363xa'}
ConfigurationServerVersion: '1.6.0.10124'
   MasterConfigurationStore: 'C:\Program Files\IVI\Data\
                              IviConfigurationStore.xml'
                 IVIRootPath: 'C:\Program Files\IVI\'
```

Modules refer to IVI-C drivers and ProgramIDs refer to IVI-COM drivers.

Logical names are associated with particular IVI drivers as defined in the IVI Configuration Store, but they do not necessarily imply that the drivers are currently installed. You can install drivers that do not have a `LogicalName` property set yet, or drivers whose `LogicalName` was removed.

Alternatively, use the Test & Measurement Tool to view the installation of IVI drivers and the setup of the IVI configuration store. Expand the `Instrument Drivers` node and click **IVI**. Click the **Software Modules** tab. (For information on the other IVI driver tabs and settings in the Test & Measurement Tool, see "IVI Configuration Store" on page 13-16.)

5 You can create an `ivi` object or a `visa` object to communicate with your instrument. For instructions on creating an IVI object, constructing an IVI configuration store, and configuring communication using an IVI-C class complaint interface, see "Reading Waveforms Using the IVI-C Class Compliant Interface" on page 13-23.

You can create a VISA object to use with different instrument types. For example, create a VISA-serial object connected to serial port `COM1` using National Instruments VISA interface.

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

If you do not get an error, the object was created successfully. If the resource name does not exist, you will get an error. Check that the resource name is correct in the vendor configuration utility. See the table in the next section.

6   Make sure you can connect to the remote host, using the `fopen` function with the object name.

```
fopen(v);
```

If you do not get an error, the connection was made successfully.

7   When you have connected, you can communicate with your device. If you have problems sending or receiving, you may need to configure communication settings. If you are using VISA, check steps listed for the interface you are using, such as TCP/IP, Serial, etc. Make sure you are using the correct instrument command. Look in the instrument's documentation to see what commands it recognizes. Verify that communication works using the vendor utility.

VISA Supported Vendor and Resource Names

When you use `instrhwinfo` to find commands to configure the interface objects, you must use valid vendor or resource names. The supported values for *vendor* are given below in parentheses after the vendor name. These are their utilities.

Vendor	Configuration Utility	Testing Connection	Debug Utlity
Agilent Technologies VISA (`agilent`)	Agilent Connection Expert (ACE)	Tools > Interactive IO	Tools > IO Monitor
National Instruments VISA (`ni`)	Measurement and Automation Explorer (MAX)	Tools > NI VISA > VISA Interactive Control	NI Spy (external application)
Tektronix VISA (`tek`)	Open Choice Instrument Manager	Open Choice Talk Listener	Open Choice Call Monitor

Hardware Support Packages

The Instrument Control Toolbox includes support packages that allow the use of certain classes of instruments. To use any of the instrument support outlined below, install the appropriate support package. See the topics listed in the table for information about the contents and installing that support package.

Support Package	Contents and Installation
Ocean Optics Spectrometers	Use to communicate with Ocean Optics USB spectrometers. Acquire data from the spectrometer and control it, including acquire a spectrum, set the integration time, and enable dark current and nonlinear spectral corrections. See "Install the Ocean Optics Spectrometers Support Package" on page 14-2.
Aardvark I2C/SPI Interface	Use the I2C or SPI interface with the Aardvark adaptor. See "Install the Total Phase Aardvark I2C/SPI Support Package" on page 14-9.
NI-845x I2C/SPI Interface	Use the I2C or SPI interface with the NI-845x adaptor. See "Install the NI-845x I2C/SPI Interface Support Package" on page 14-8.
NI-SCOPE Oscilloscopes	Use to communicate with NI-SCOPE oscilloscopes. Acquire waveform data from the oscilloscope and control it. See "Install the NI-SCOPE Support Package" on page 14-4.
NI-FGEN Function Generators	Use to communicate with NI-FGEN function generators. Control and configure the function generator, and perform tasks such as generating sine waves. See "Install the NI-FGEN Support Package" on page 14-5.
NI-DCPower Power Supplies	Use to communicate with NI-DCPower power supplies. Control and take digital measurements from a power supply, such as the NI PXI 4011 triple-output programmable DC power supply.

Support Package	Contents and Installation
	See "Install the NI-DCPower Support Package" on page 14-6.
NI-DMM Digital Multimeters	Use to communicate with NI-DMM digital multimeters. Control and take measurements from a digital multimeter, such as measuring voltage or resistance. See "Install the NI-DMM Support Package" on page 14-7.
NI-Switch Hardware	Use to communicate with NI-Switch instruments. For example, control a relay box such as the NI PXI-2586 10-channel relay switch. See "Install the NI-Switch Support Package" on page 14-10.
NI VISA and ICP Interfaces	Use the Quick Control Oscilloscope and Quick Control Function Generator interfaces to communicate with and control oscilloscopes and function generators. See "Install the National Instruments VISA and ICP Interfaces Support Package" on page 14-11.

Deploying Standalone Applications with Instrument Control Toolbox

In this section...
"Tips for both interface based communication and driver-based communication" on page 20-49
"Tips for interface based communication" on page 20-49
"Tips for driver based communication" on page 20-49
"Hardware Support packages" on page 20-52

This topic contains tips for deploying standalone applications with MATLAB Compiler (deploytool or mcc) and Instrument Control Toolbox. Refer to these tips when creating standalone applications that use functionality from Instrument Control Toolbox.

Tips for both interface based communication and driver-based communication

- Device identifiers/resource names should not be hard-coded, as instrument resource names are likely to be different on other machines.
- A best practice is to use `instrhwinfo` and query the return output in your MATLAB code that you intend to deploy.

Tips for interface based communication

For direct interface based communication using I2C, SPI, GPIB, and VISA, on the deployment machine, install all required third-party drivers separately from the deployed application.

Tips for driver based communication

In addition to your MATLAB code, your deployed standalone application package should include files required by your application:

- Include MATLAB Instrument Driver (MDD file) in your standalone application project from `deploytool` or by passing a `-a` flag to `mcc` when compiling your MATLAB code.

IVI-C

Include the following in your deployed standalone application package:

- MATLAB Instrument Driver (MDD file)
- For 64-bit applications, MATLAB prototype and thunk files
- For 32-bit applications, MATLAB prototype file

The location of generated prototype and thunk files can be obtained from the result of executing:

```
sprintf('%s',[tempdir 'ICTDeploymentFiles'])
```

On the machine where you deploy your standalone application:

- All third-party drivers and dependencies need to be installed separately from the deployed standalone application.
- To reduce runtime unknowns, install the same version of IVI-C driver on the deployment system as used on the development system.
- To reduce runtime unknowns, install the same version of VISA driver libraries on the deployment system as used on the development system.

Note: To troubleshoot vendor driver installation issues, it is recommended that the deployed application provide a way to simulate connection to the hardware by instantiating the driver with `'optionstring', 'simulate=true'` as arguments for `icdevice`. This will help narrow down the root cause of deployment issues to vendor driver installation issue, or hardware issues.

Quick Control Interfaces

If you are not using the default SCPI-based drivers for Quick Control Oscilloscope (`'tektronix'`) and Quick Control Function Generator (`'Agilent332x0_SCPI'`), and are instead using an IVI-C driver, include the following in your deployed standalone application package:

- For 64-bit applications, MATLAB prototype and thunk files for IVIScope or IVIFGen
- For 32-bit applications, MATLAB prototype file for IVIScope or IVIFGen

The location of generated prototype and thunk files can be obtained from the result of executing:

```
sprintf('%s',[tempdir 'ICTDeploymentFiles'])
```

On the machine where you deploy your standalone application:

- If required, all third-party drivers and dependencies need to be installed separately from the deployed standalone application.
- To reduce runtime unknowns, install the same version of IVI-C driver on the deployment system as used on the development system.
- To reduce runtime unknowns, install the same version of VISA driver libraries on the deployment system as used on the development system.

Generic MDD

For use with the generic MDD, include the following in your deployed standalone application package:

- MATLAB Instrument Driver (MDD file)

If your MDD uses LOADLIBRARY to interface with a C shared library, include:

- For 64-bit applications, MATLAB prototype and thunk files for the C shared library
- For 32-bit applications, MATLAB prototype file for the C shared library
- MATLAB prototype and thunk files for a C shared library can be generated on a development machine (with a supported C compiler) by using LOADLIBRARY command.

On the machine where you deploy your standalone application:

- If required, all third-party drivers and dependencies need to be installed separately from the deployed standalone application.

IVI-COM

IVI-COM is only supported with 32-bit MATLAB. If you generate a standalone executable, you will generate an x86 executable that will require you to install the x86 versions of the drivers. It is strongly recommended that you use the IVI-C driver in your MATLAB code.

For use with IVI-COM, include the following in your deployed standalone application package:

- MATLAB Instrument Driver (MDD file)

On the machine where you deploy your standalone application:

- All third-party drivers and dependencies need to be installed separately from the deployed standalone application.
- To reduce runtime unknowns, install the same version of IVI-COM driver on the deployment system as used on the development system.
- To reduce runtime unknowns, install the same version of VISA driver libraries on the deployment system as used on the development system.

Note: To troubleshoot vendor driver installation issues, it is recommended that the deployed application provide users a way to simulate connection to the hardware by instantiating the driver with `'optionstring', 'simulate=true'` as arguments for `icdevice`. This will help narrow down the root cause of deployment issues to vendor driver installation issue, or hardware issues.

Hardware Support packages

For more info on deploying standalone applications which use functionality installed as a support package:

```
web(fullfile(docroot, 'compiler/manage-support-packages.html'))
```

Contact MathWorks and Use the instrsupport Function

If you need support from MathWorks, visit our Web site at http://www.mathworks.com/support/.

Before contacting MathWorks, run the `instrsupport` function. This function returns diagnostic information such as:

- The versions of MathWorks products you are using
- Your MATLAB path
- The characteristics of your hardware
- Information about your adaptors

The output from `instrsupport` is automatically saved to a text file, `instrsupport.txt`, which you can use to help troubleshoot your problem.

To have MATLAB generate this file for you, type

```
instrsupport
```

Using the Instrument Control Toolbox Block Library

The Instrument Control Toolbox software includes a Simulink software interface called the Instrument Control Toolbox block library. You can use the blocks of this library in a Simulink model to communicate with an instrument.

Overview

The Instrument Control Toolbox software includes a Simulink software interface called the Instrument Control Toolbox block library. You can use the blocks of this library in a Simulink model to communicate with an instrument.

The topics in this section describe how to use the Instrument Control Toolbox block library. The block library consists of these blocks:

- **Query Instrument** — Query the instrument for data.
- **Serial Configuration** — Configure a serial port to send an receive data.
- **Serial Receive** — Receive data over a serial network.
- **Serial Send** — Send data over a serial network.
- **TCPIP Receive** — Receive data over a TCP/IP network.
- **TCPIP Send** — Send data over a TCP/IP network.
- **To Instrument** — Send data to the instrument.
- **UDP Receive** — Receive data over an UDP network.
- **UDP Send** — Send data over and 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 the Simulink Getting Started Guide in the Simulink documentation to better understand its functionality.

For more detailed information about the blocks in the Instrument Control Toolbox block library, see the blocks documentation. For examples of using the Instrument Control Toolbox block library to build models to send and receive data, see "Building Simulink Models to Transmit Data" on page 21-7.

Opening the Instrument Control Block Library

In this section...

Using the instrumentlib Command from MATLAB

To open the Instrument Control block library, enter

```
instrumentlib
```

in the MATLAB Command Window. MATLAB displays the contents of the library in a separate window.
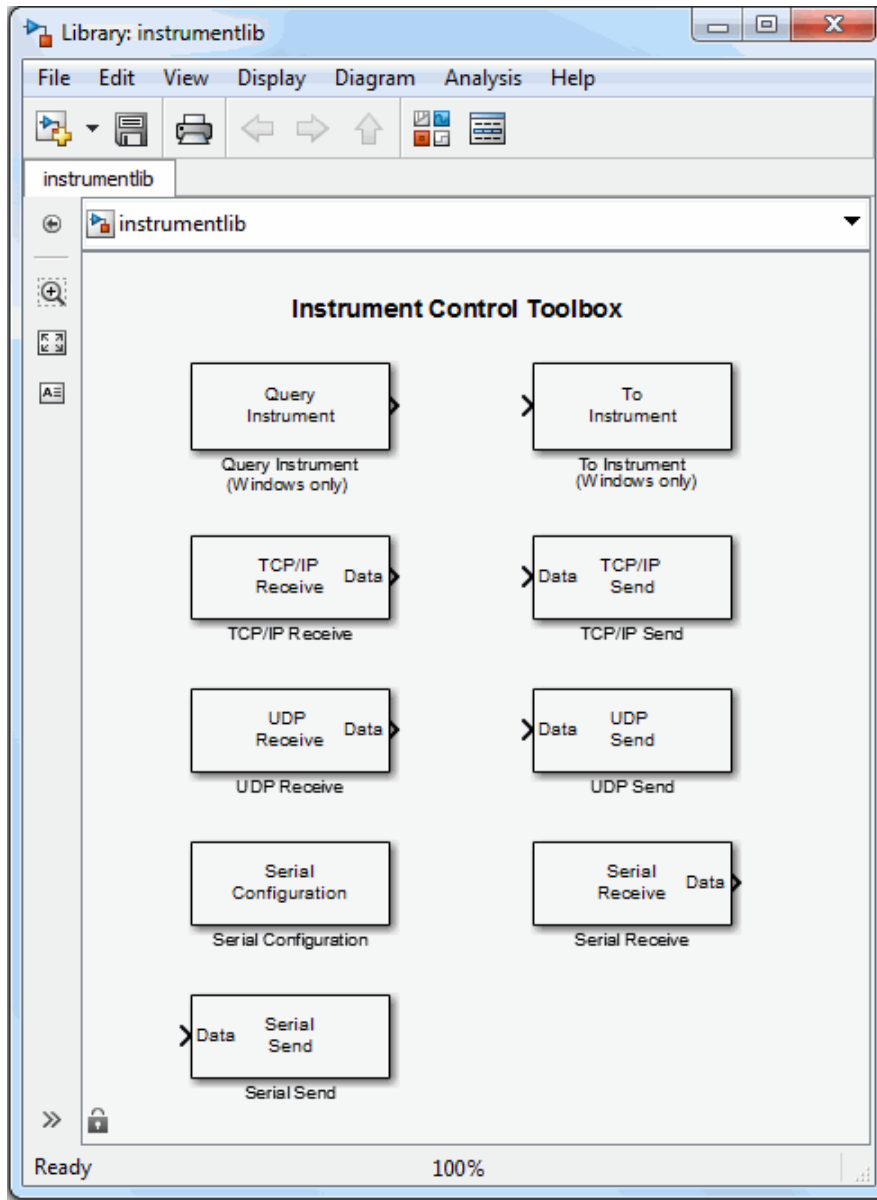
Using the Simulink Library Browser

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
```

in the MATLAB Command Window. MATLAB opens the browser window. The left pane lists available block libraries, with the basic Simulink library listed first, followed by other libraries alphabetical order. To open the Instrument Control Toolbox block library, click its icon.
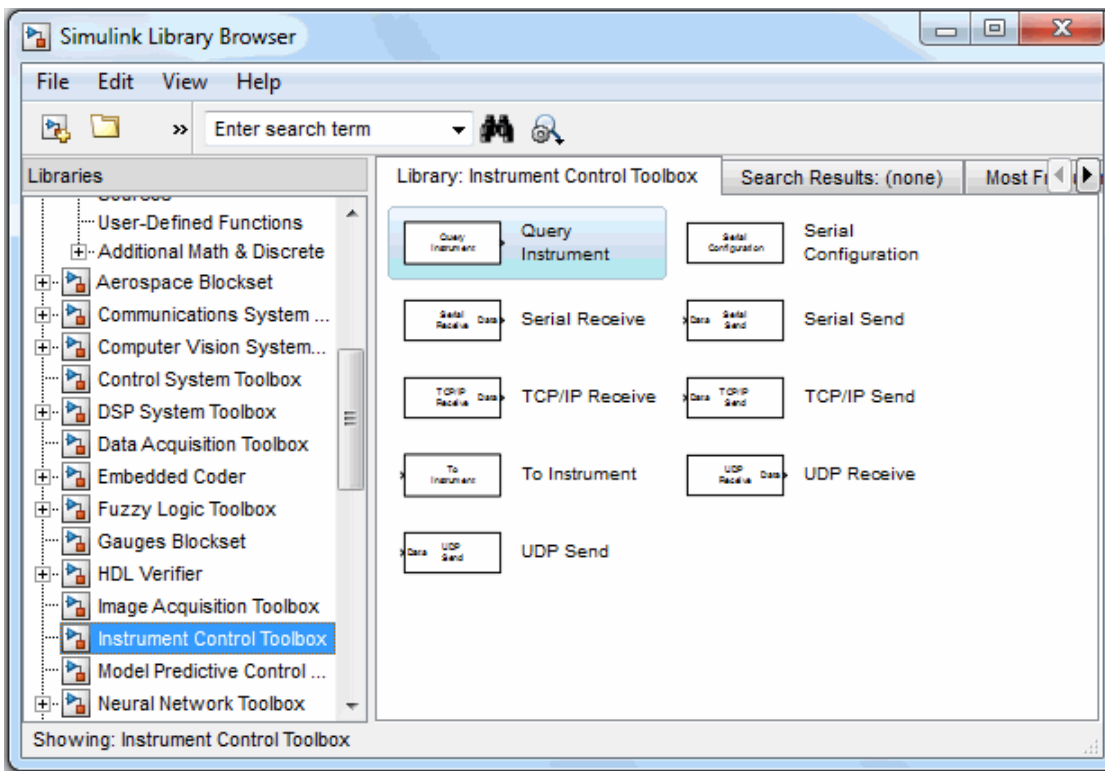
Simulink Library Browser

Simulink loads the library and displays the blocks in the library.

Building Simulink Models to Transmit Data

In this section...

Sending and Receiving Data Through a Serial Port Loopback

This section provides an example that builds a simple model using the Instrument Control Toolbox blocks in conjunction with other blocks in the Simulink library. The example illustrates how to send data to a simple loopback device connected to the computer's 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 21-7
- "Step 2: Create a New Model" on page 21-8
- "Step 3: Drag the Instrument Control Toolbox Blocks into the Model" on page 21-8
- "Step 4: Drag Other Blocks to Complete the Model" on page 21-10
- "Step 5: Connect the Blocks" on page 21-12
- "Step 6: Specify the Block Parameter Values" on page 21-13
- "Step 7: Specify the Block Priority" on page 21-15
- "Step 8: Run the Simulation" on page 21-16

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 in the browser.

To start the Simulink Library Browser, enter

```
simulink
```

in the MATLAB Command Window. The left pane in the Simulink Library Browser lists the available block libraries. To open the Instrument Control Toolbox block library, click its entry icon.

Step 2: Create a New Model

To use a block, add it to an existing model or create a new model.

For this example, create a new model by clicking the **New model** button on the toolbar.
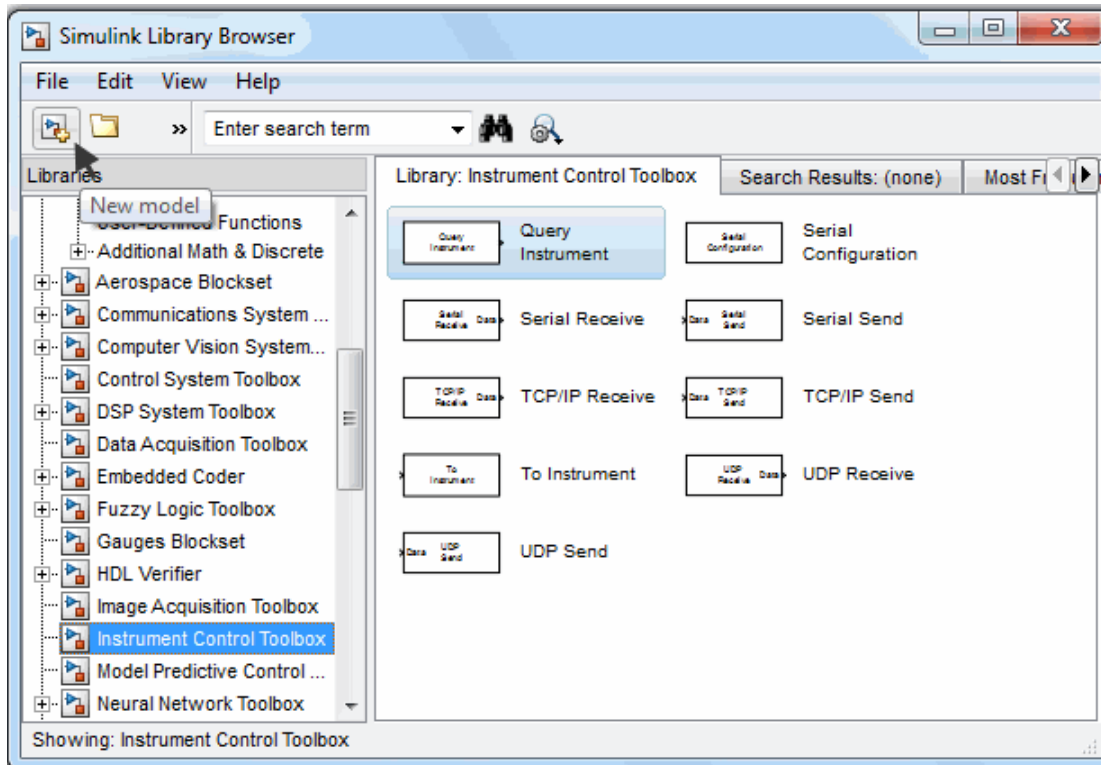
You can also select **File > New** in the Simulink Library Browser. Simulink opens an empty model in the Simulink Editor. Name the new model using the **Save** option.

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 Simulink Editor. For this example, you need one instance of the To Instrument and the Query Instrument blocks in your model.

Note: The To Instrument block can be used with these interfaces: VISA, GPIB, Serial, TCP/IP, and UDP. It is not supported on these interfaces: SPI, I2C, and Bluetooth.
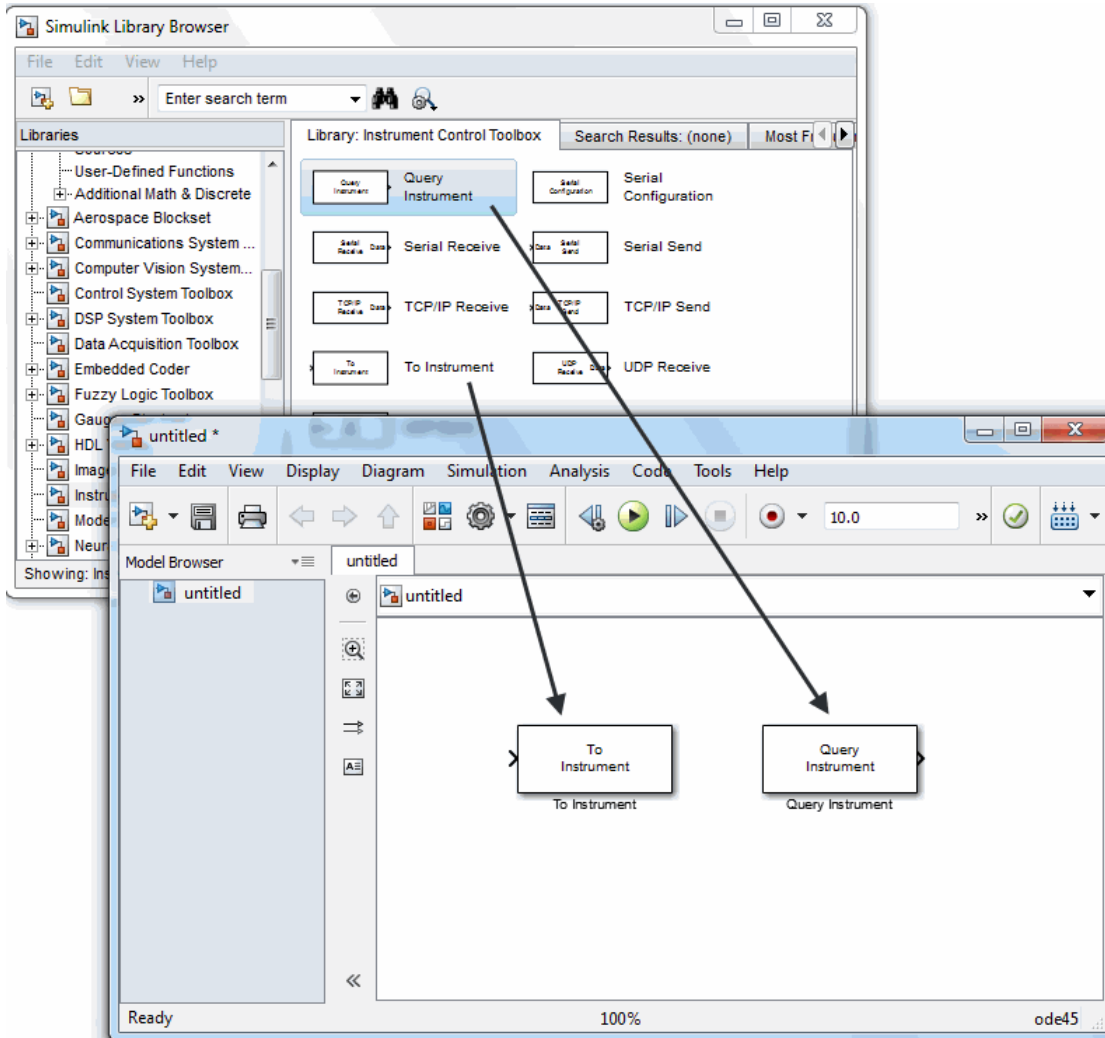
Drag Instrument Control Toolbox Blocks into Model

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. Access the block by expanding the Simulink node in the browser tree, and clicking the **Sources** library entry. From the blocks in the right pane, drag the Constant block into the Simulink Editor and place it to the left of the To Instrument block.
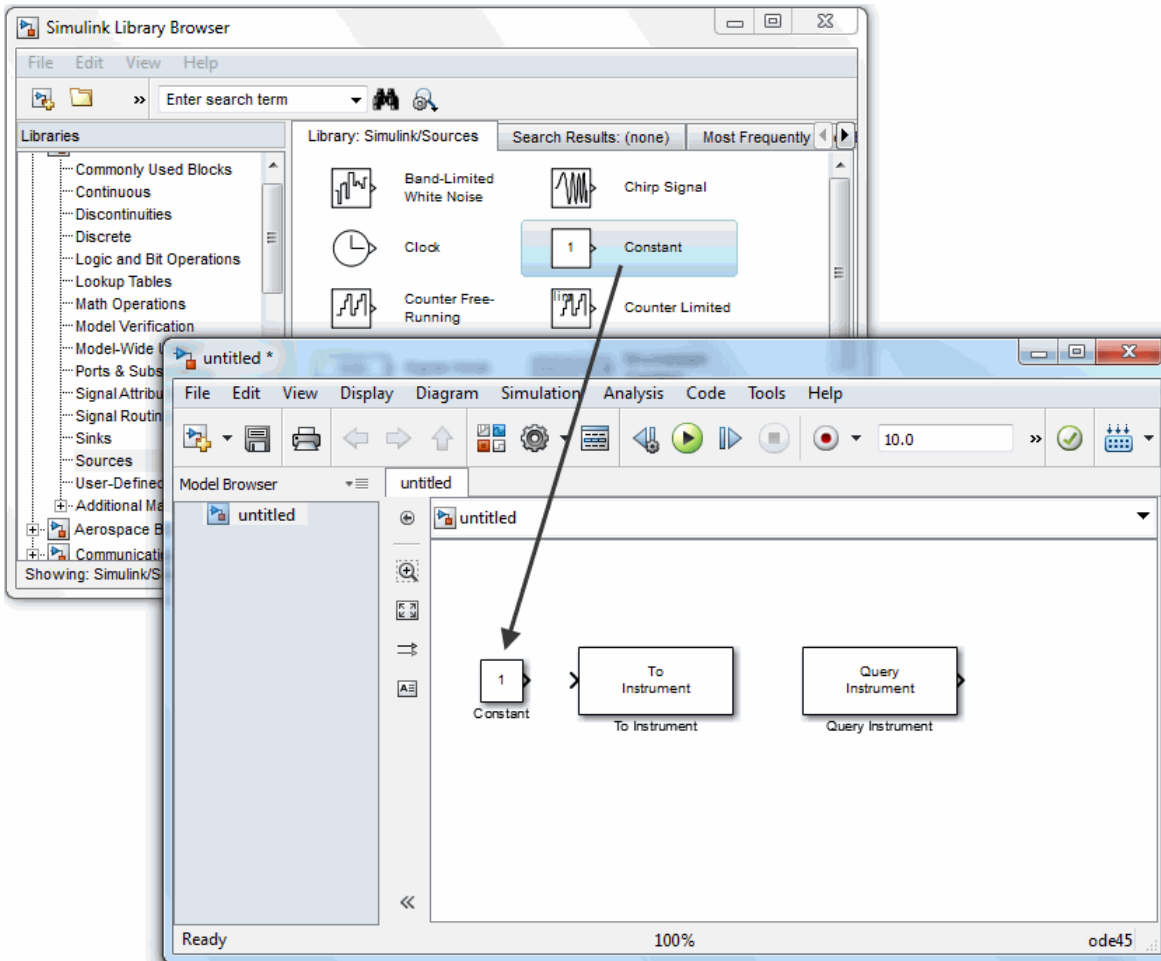
Drag Constant Block to the Model Window

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 Simulink Editor and place it to the right of the Query Instrument block.

Step 5: Connect the Blocks

Make a connection between the Constant block and the To Instrument block. A quick way to make the connection is to select the Constant block, press and hold the **Ctrl** key, and then click the To Instrument block.

In the same way, make the connection between the output port of the Query Instrument block and the input port of the `Display` block.
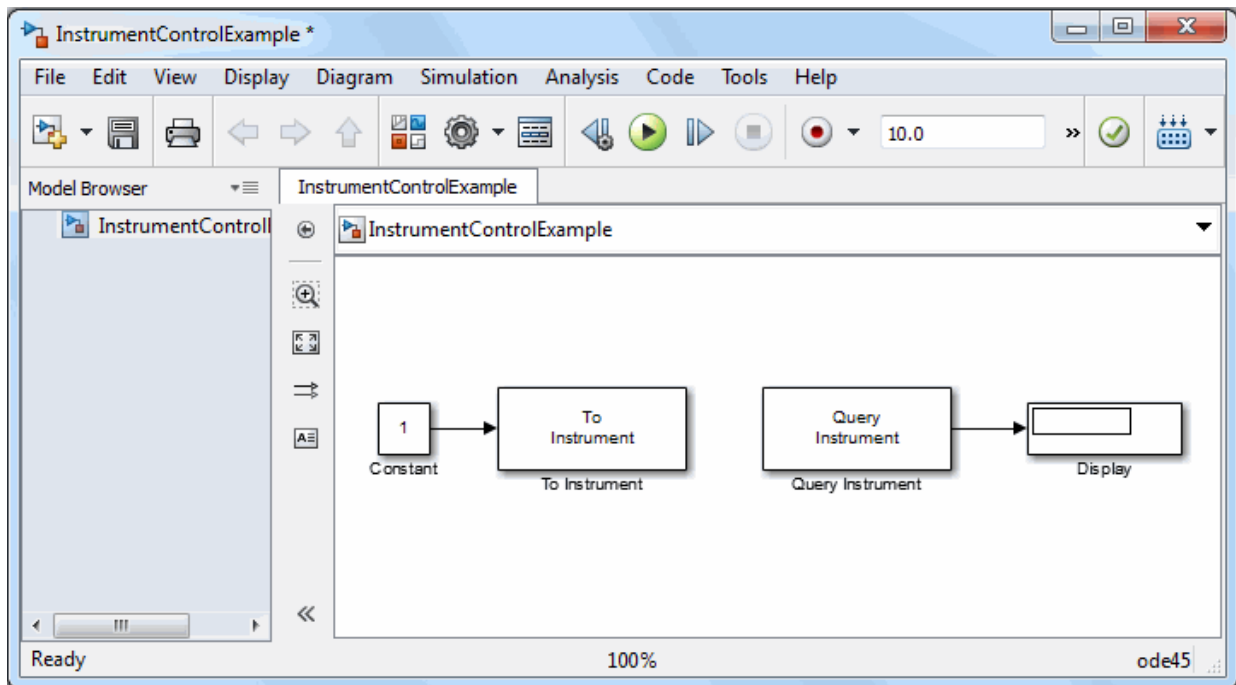
Note: The two blocks do not directly connect together within the model. The only communication between them is through the instrument, which is the loopback connected to the COM1 serial port. Because there is no direct connection between these two blocks, you must consider their timing when running the 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

Set parameters for the blocks in your model by double-clicking the block.

Double-click the To Instrument block to open its parameters dialog box. Accept the default values configured for this block, so you do not need to modify any of the values.
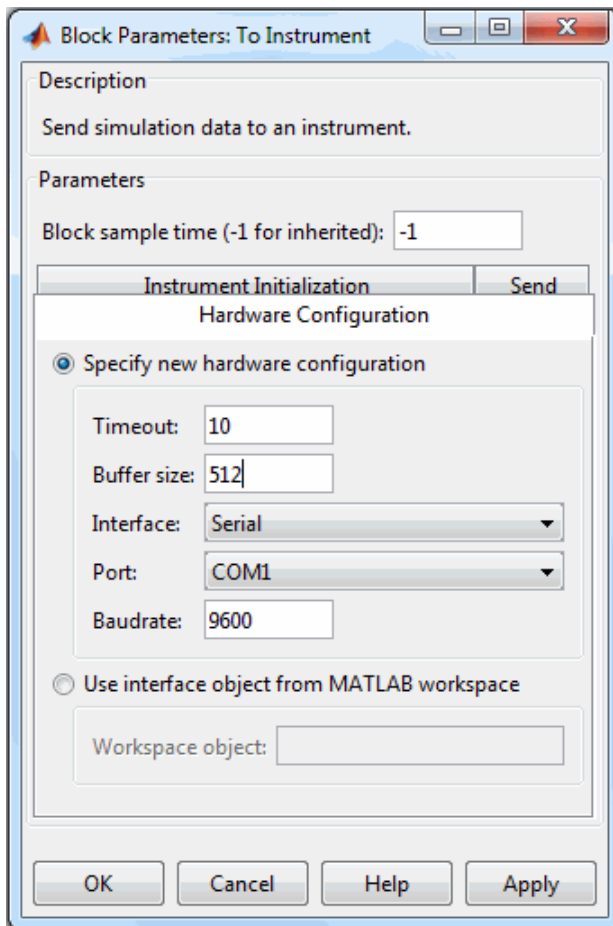
Click **OK** to close the dialog box.

Double-click the Query Instrument block to open its parameters dialog box. Make sure that the values on the **Hardware Configuration** tab match the Hardware values on the To Instrument block.
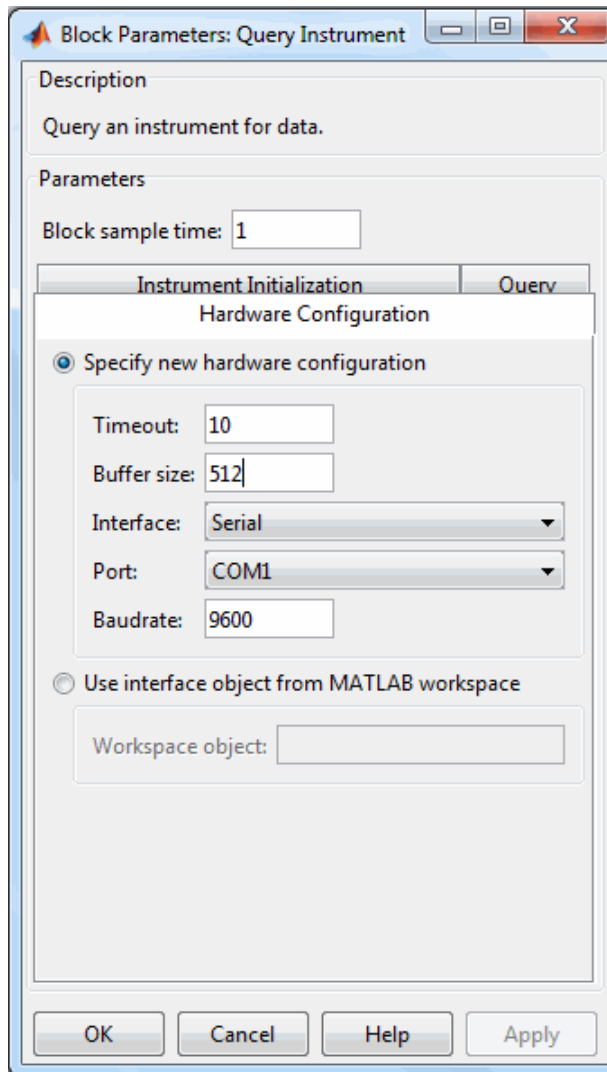
The model uses the default values on the **Instrument Initialization** and **Query** tabs of this block, so you do not need to modify any of their values.

Click **OK** to apply any changes and close the dialog box.

Double-click the Constant block to open its parameters dialog box. Change the Constant value to the value you want to send to the instrument. For this example, change:

1 **Constant value** to 25.
2 **Sample time** to 1.
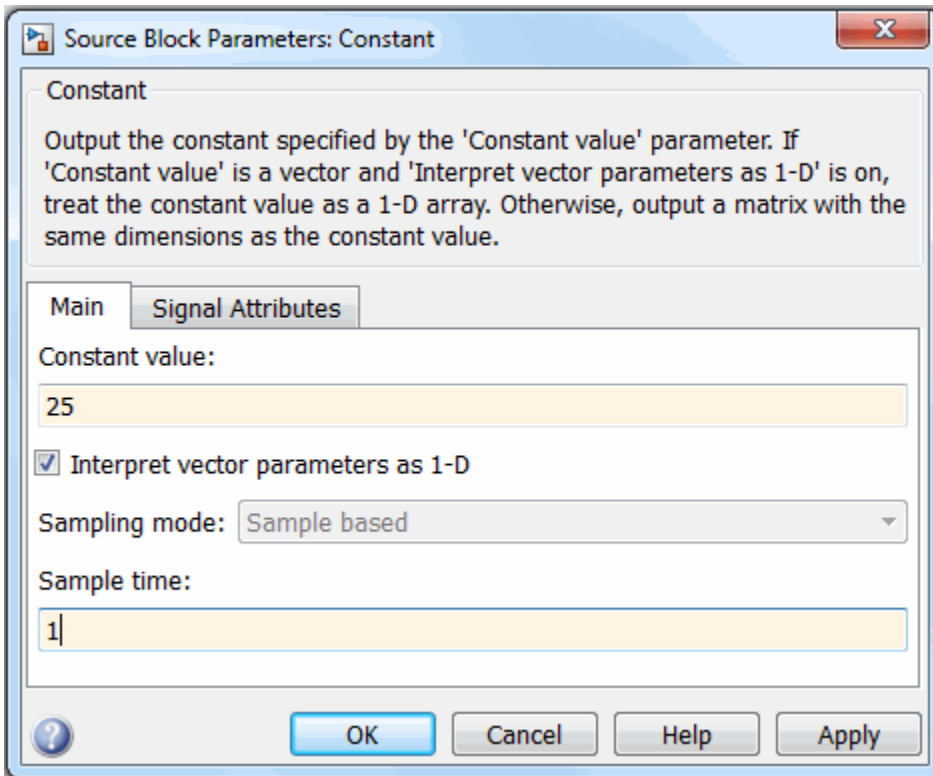
Click **OK**.

For the Display block, you can use its default parameters.

Step 7: Specify the Block Priority

The block with the lowest number gets the highest priority. In the Simulink Editor, right-click a block and select **Properties**. Enter the priority number in the **Priority**

field in the Block Properties dialog box. To ensure that the To Instrument block first completes writing data to the loopback before the Query Instrument block reads it, set the priority of the To Instrument block to 1 and the Query Instrument block to 2.

Caution It is essential to set the correct priority for the blocks in your model. Otherwise you may see unexpected results.

To understand more about block priorities, see the Simulink documentation.

Step 8: Run the Simulation

To run the simulation, click the green **Run** button on the Simulink Editor toolbar. You can use toolbar options to specify how long to run the simulation and to stop a running simulation. You can also start the simulation by selecting **Simulation > Run**.
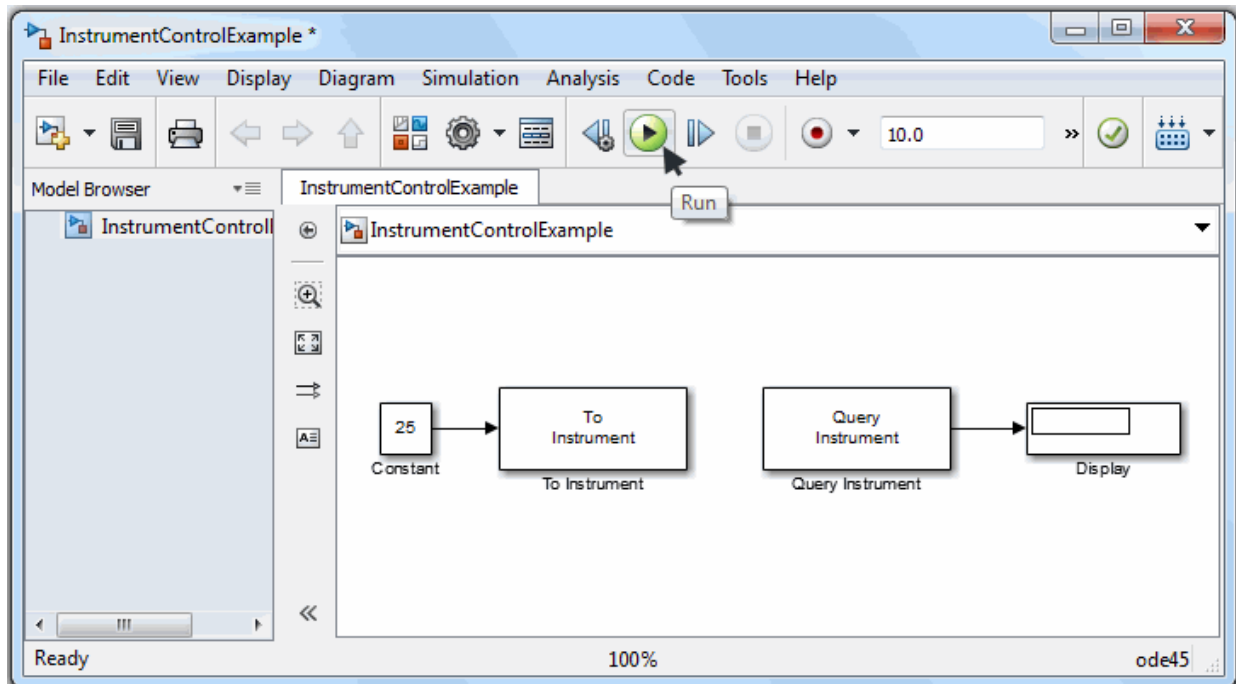
When you run the simulation, the constant value you specified (25) is written to the instrument (the serial loopback), received from the instrument, and shown in the `Display` block.

While the simulation is running, the status bar at the bottom of the Simulink Editor updates the progress of the simulation.

Sending and Receiving Data Over a TCP/IP Network

This example builds a simple model using the Instrument Control blocks in the block library in conjunction with other blocks in the Simulink library. This example also illustrates how to send data to an echo server using TCP/IP and to read that data back into your model.

You will create an echo server on your machine that simulates sending a signal to the TCP/IP send block and echo the result back to the Send block to send data, 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 21-17
- "Step 2: Open the Block Library" on page 21-18
- "Step 3: Create a New Model" on page 21-18
- "Step 4: Drag the Instrument Control Toolbox Blocks into the Model" on page 21-19
- "Step 5: Drag the Sine Wave and Scope Blocks to Complete the Model" on page 21-20
- "Step 6: Connect the Blocks" on page 21-25
- "Step 7: Specify the Block Parameter Values" on page 21-25
- "Step 8: Specify the Block Priorities" on page 21-28
- "Step 9: Run the Simulation" on page 21-28
- "Step 10: View the Result" on page 21-29

Step 1: Create an Echo Server

Open a port on your computer to work as an echo server that you can use to send and receive signals via TCP/IP. To create an echo server, type this command in MATLAB:

```
echotcpip('on', 50000)
```

Port 50000 opens 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 in the browser.

To start the Simulink Library Browser, enter

```
simulink
```

in the MATLAB Command Window. 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.

Step 3: Create a New Model

To use a block, add it to an existing model or create a new model.

For this example, create a new model by clicking the **New model** button on the toolbar.
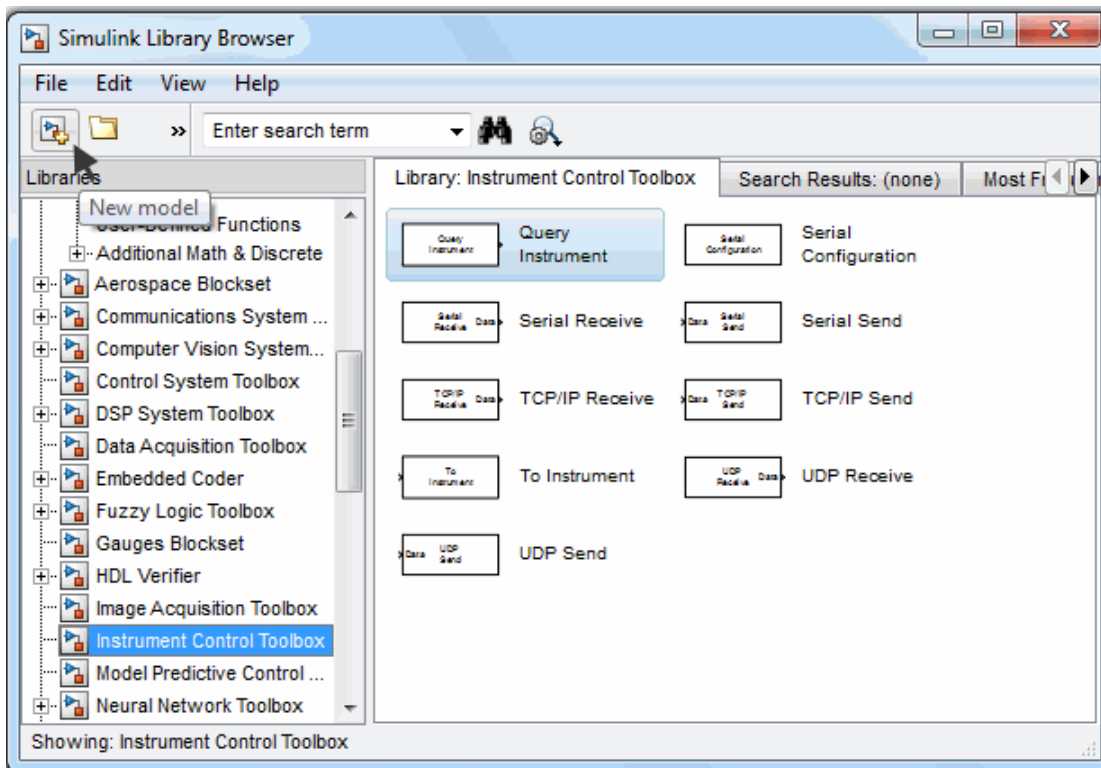
You can also select **File > New** in the Simulink Library Browser. Simulink opens an empty model in the Simulink Editor. Name the new model using the **Save** option.

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 Simulink Editor. For this model, you need one instance of the TCP/IP Send and the TCP/IP Receive blocks in your model.
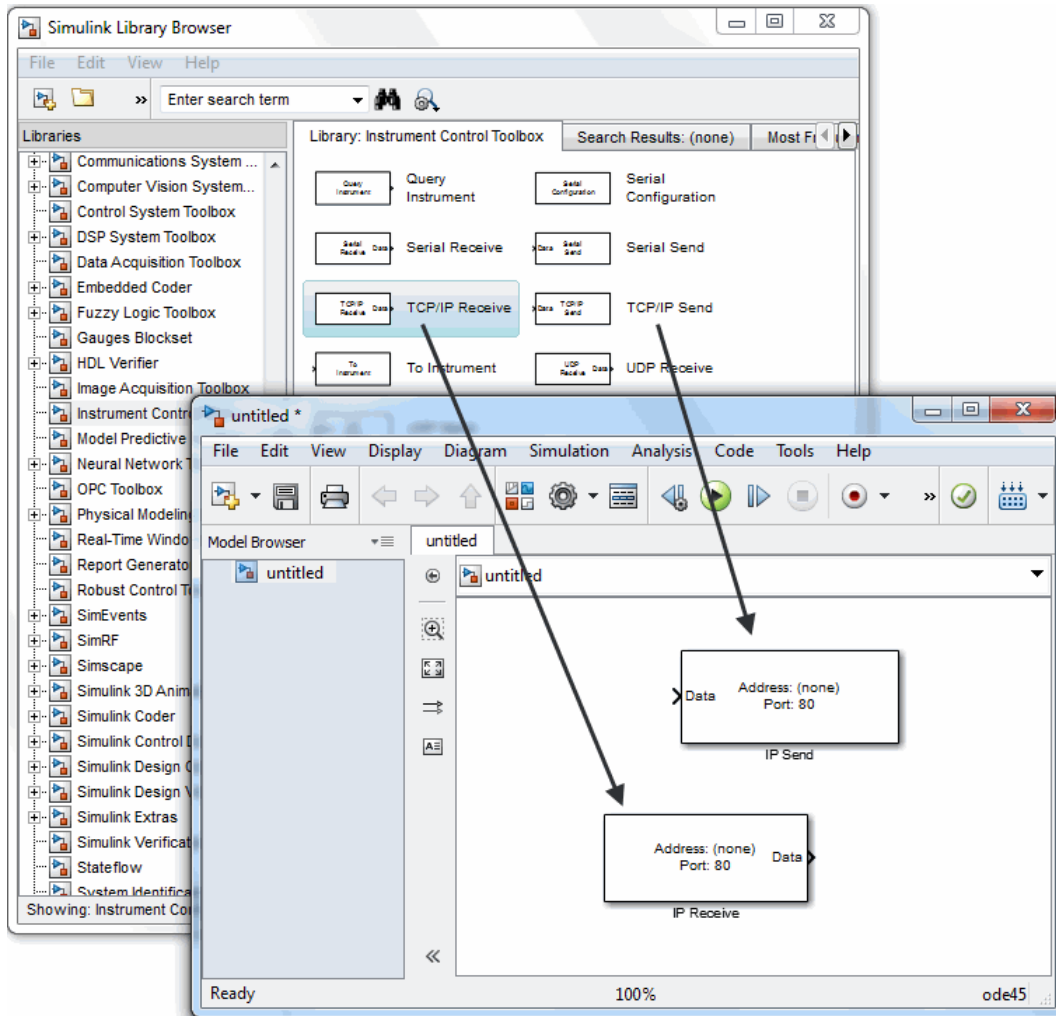
Drag Instrument Control Toolbox Blocks into the Model

Step 5: Drag the Sine Wave and Scope Blocks to Complete the Model

This example requires two more blocks. One block displays the data received from the 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 the Sources library entry. From the blocks in the right pane, drag the Sine Wave block into the model and place it to the left of the TCP/IP Send block.

Drag Sine Wave Block to the Model
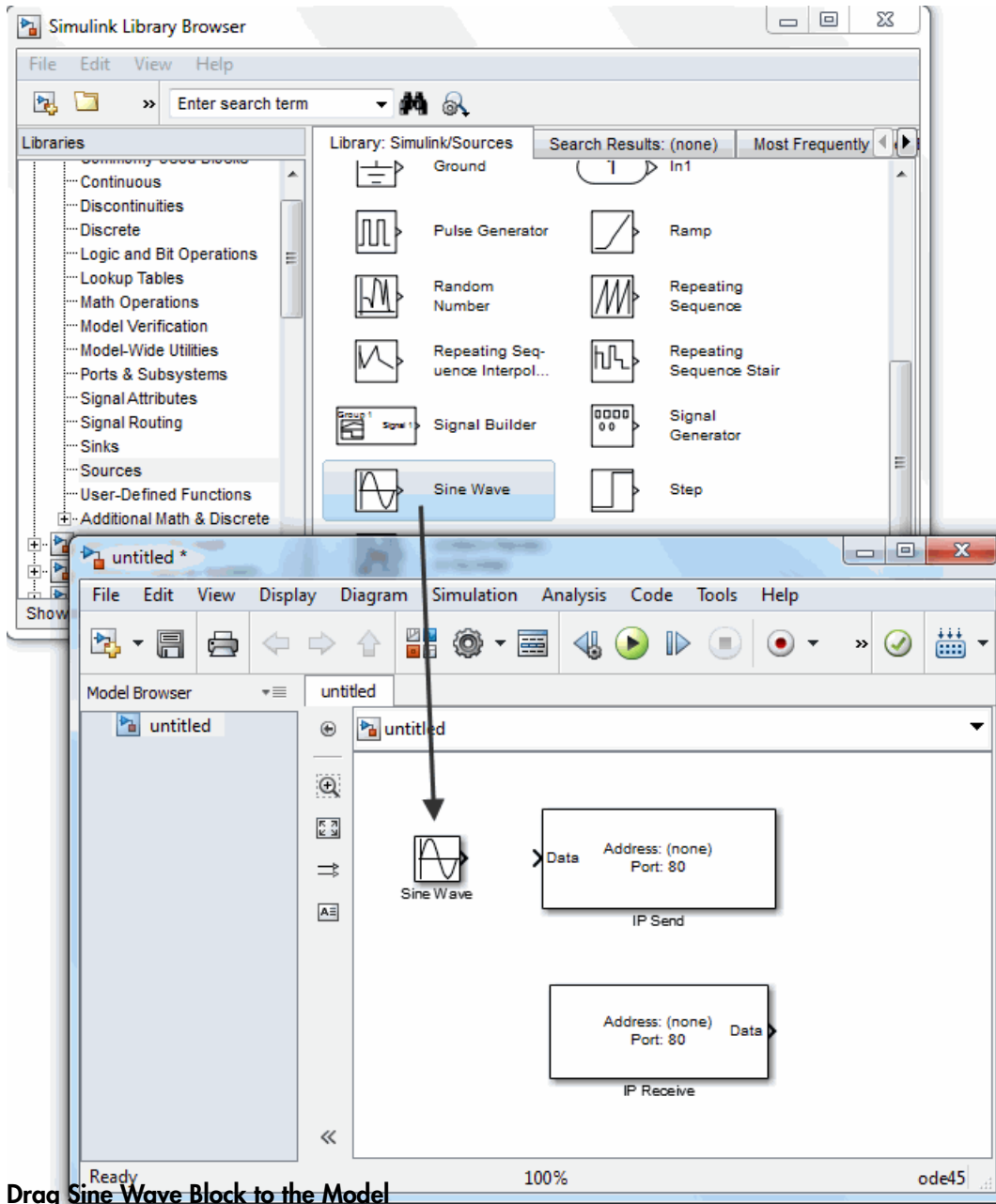
To display the data received by the TCP/IP Receive block, use the `Scope` block. To access this block, click the Sinks library entry in the expanded Simulink node in the browser tree. From the blocks in the right pane, drag the Display block into the model and place it to the right of the TCP/IP Receive block.
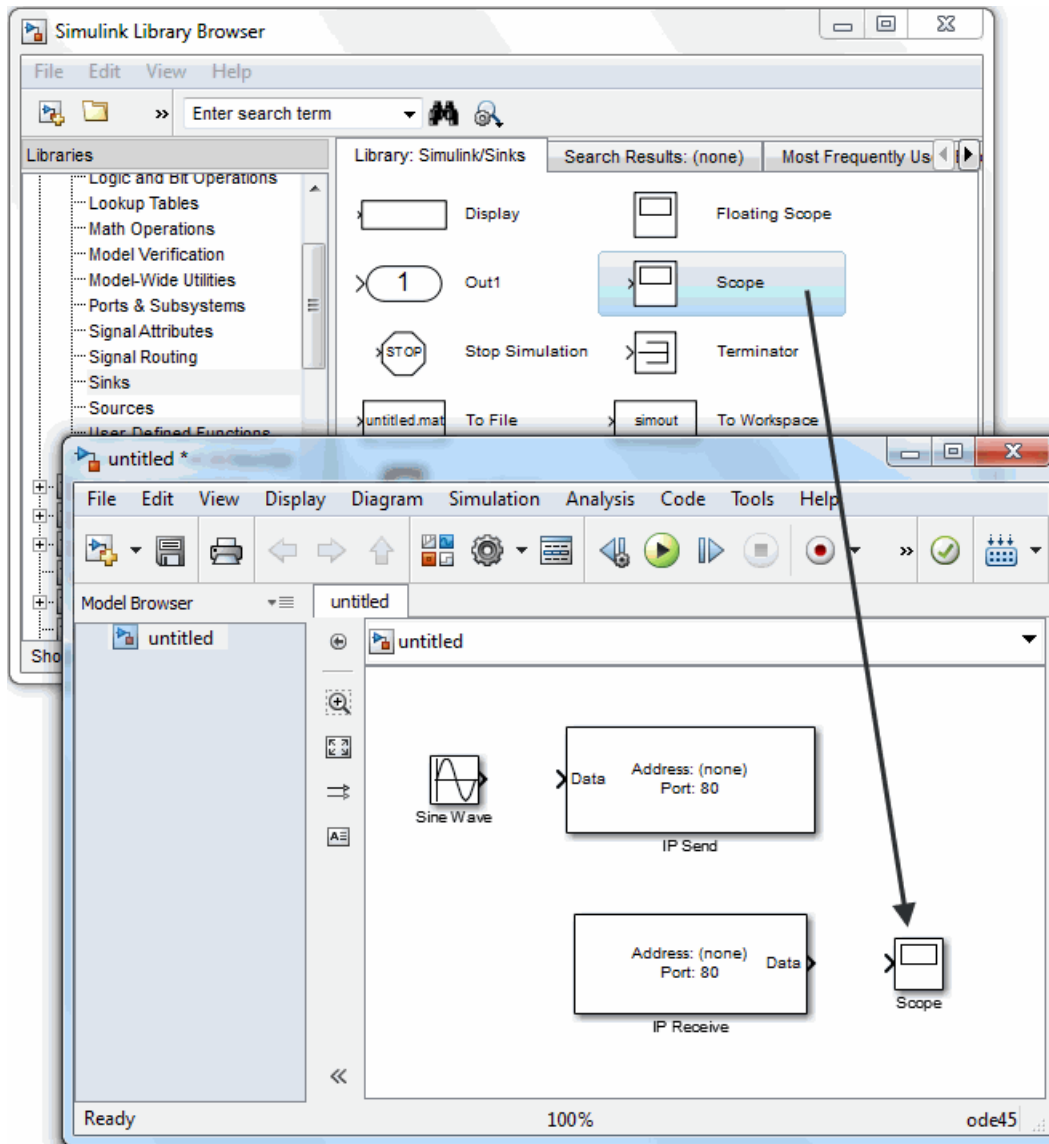
Drag Scope Block to the Model

Step 6: Connect the Blocks

Make a connection between the Sine Wave block and the TCP/IP Send block. A quick way to make the connection is to select the Sine Wave block, press and hold the **Ctrl** key, and then click the TCP/IP Send block. In the same way, make the connection between the output port of the TCP/IP Receive block and the input port of the Scope block.
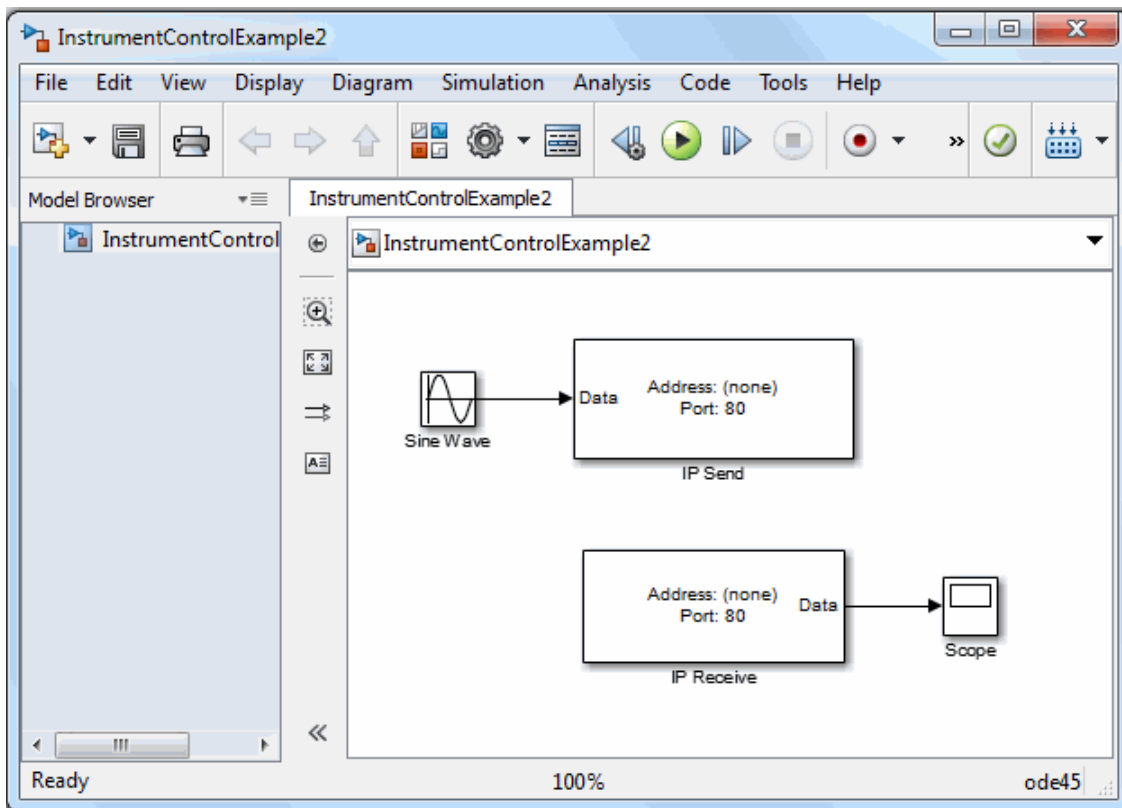
Step 7: Specify the Block Parameter Values

Set parameters for the blocks in your model by double-clicking the block.

Configure the Send Block

Double-click the TCP/IP Send block to open its parameters dialog box. Set the **Remote address** field to `localhost` and the **Port** field to `50000`, since that is the address you set the echo server to when you started it.
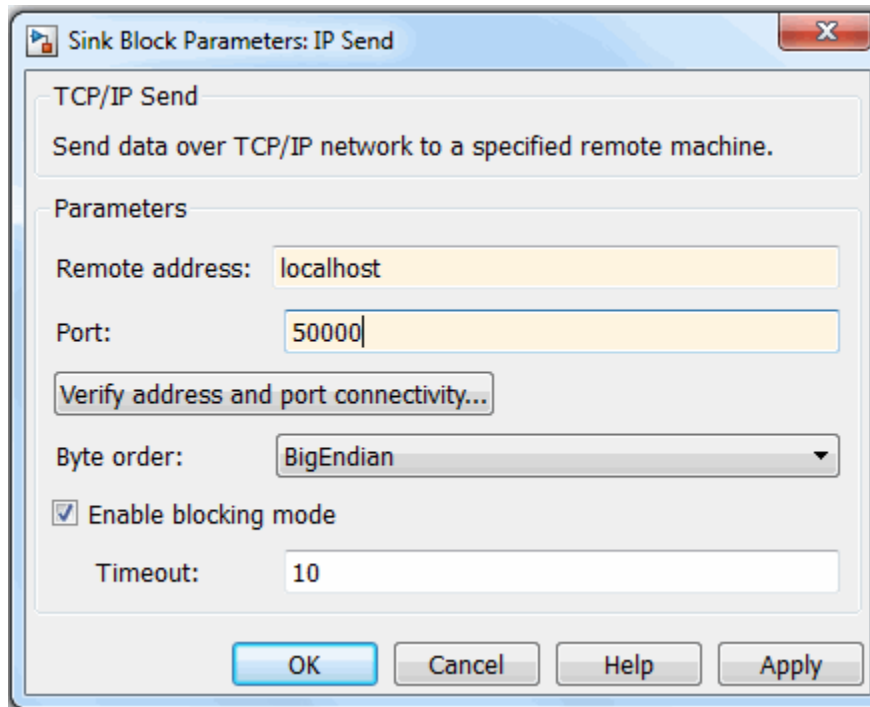
Click **Apply** and then **OK**.

Configure the Receive Block

Double-click the Receive block to open its parameters dialog box. Set the **Remote address** field to `localhost` and the **Port** field to `50000`. The **Block sample time** field is set to `0.01` by default. The block sample time here must match the one in the Sine Wave block, so confirm that they are both set to `0.01`.
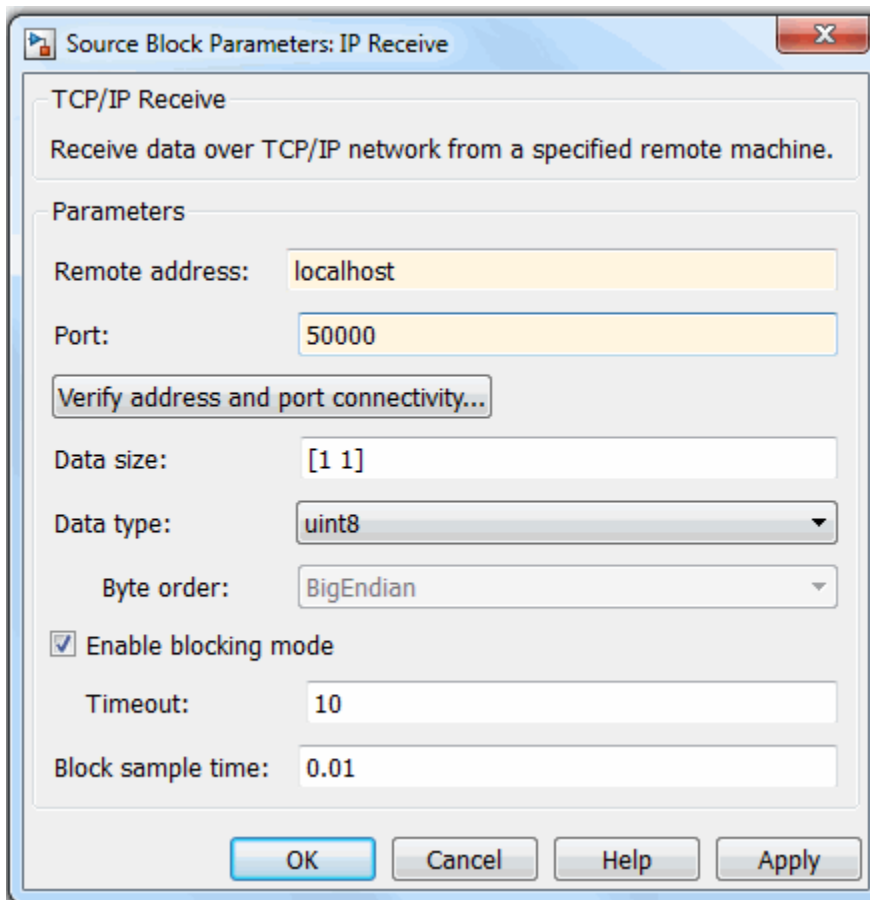
Click **OK**.

Configure the Sine Wave Block

Double-click the Sine Wave block to open its parameters dialog box. Set the **Sample time** field to `0.01`.

Click **OK**.

Step 8: Specify the Block Priorities

To run the simulation correctly, specify the order in which Simulink should process the blocks. Right-click the block and select **Properties**. Enter the priority number in the **Priority** field. In this case, set the priority of **TCP/IP Send** to 1 and **TCP/IP Receive** to 2.

Caution It is essential to set the correct priority for the blocks in your model. Otherwise, you may see unexpected results.

Read the Simulink documentation to understand more about block priorities.

You also need to set two parameters on the model. In the Simulink Editor, select **Simulation > Model Configuration Parameters**. In the Configuration Parameters dialog box, set the **Type** field to `Fixed-step` and set the **Solver** field to `discrete (no continuous states)`.
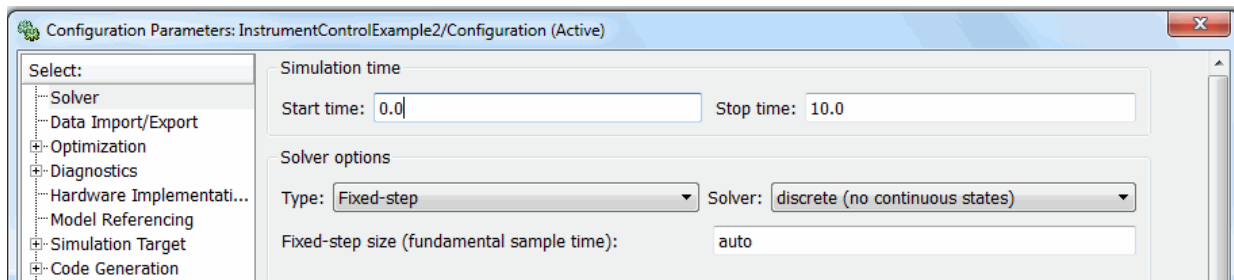
Step 9: Run the Simulation

To run the simulation, click the green **Run** button on the Simulink Editor toolbar. You can use toolbar options to specify how long to run the simulation and to stop it. You can also start the simulation by selecting **Simulation > Run**.
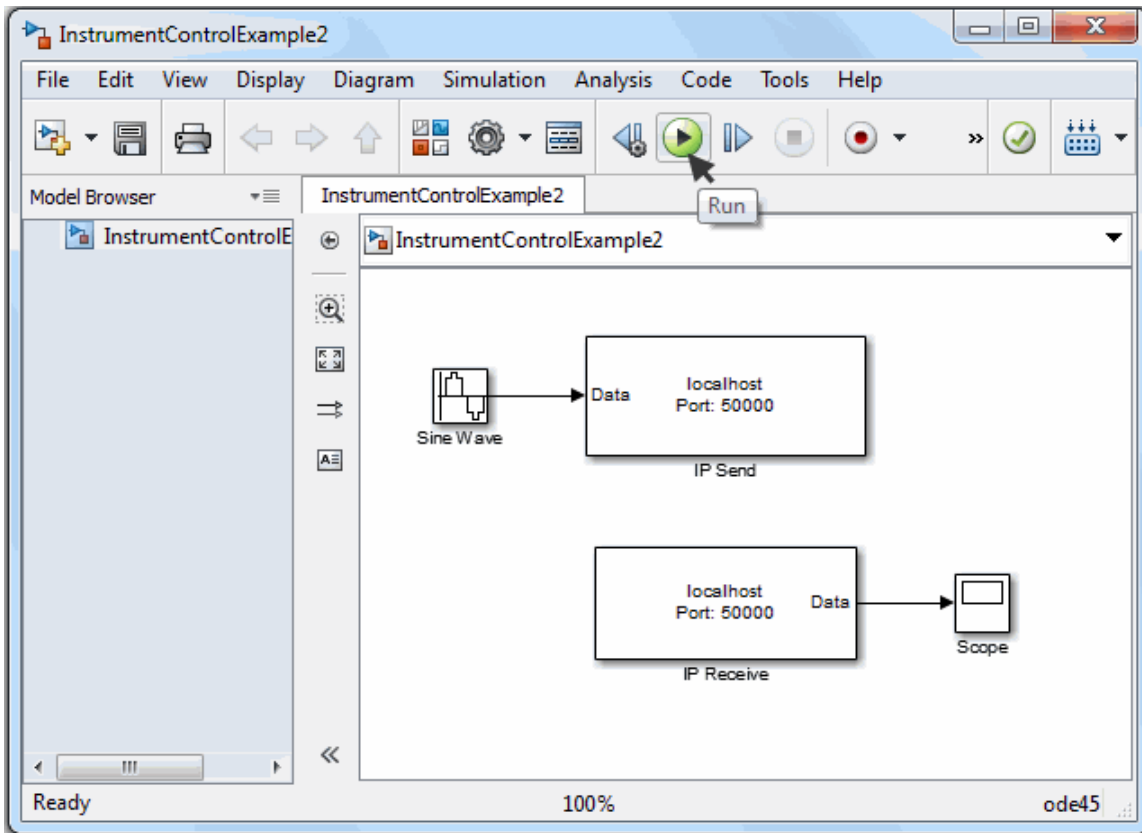
While the simulation runs, the status bar at the bottom of the Simulink Editor updates the progress of the simulation.

Step 10: View the Result

Double-click the Scope block to view the signal on a graph as it is received by the TCP/IP Receive block.
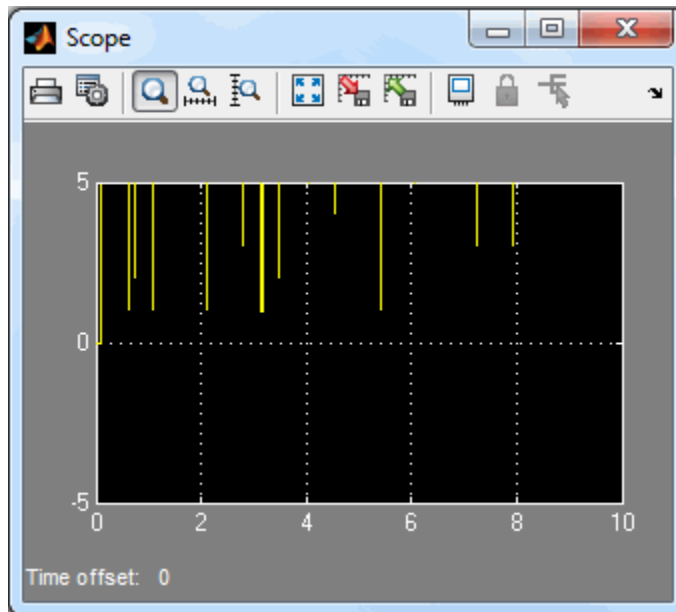
For more information about Instrument Control Toolbox blocks, see the blocks reference documentation.

Functions — Alphabetical List

Test and Measurement Tool

Control oscilloscopes and other instruments

Description

The Test & Measurement Tool enables you to configure and control resources, such as instruments, serial devices, drivers, and interfaces, through Instrument Control Toolbox without having to write a MATLAB script.

Use the Test & Measurement Tool to manage your session with the toolbox. You can:

- Detect available hardware and drivers.
- Connect to an instrument or device.
- Configure instrument or device settings.
- Read and write data.
- Automatically generate the MATLAB script.
- Visualize acquired data.
- Export acquired data to the MATLAB workspace.

Open the Test and Measurement Tool App

- MATLAB Toolstrip: On the **Apps** tab, under **Test and Measurement**, click the app icon.
- MATLAB command prompt: Enter `tmtool`.

Examples

Programmatic Use

`tmtool` opens the Test & Measurement Tool, which enables you to configure and control resources, such as instruments, serial devices, drivers, and interfaces, accessible through the Instrument Control Toolbox.

See Also

Apps

Functions

Introduced before R2006a

add

Add entry to IVI configuration store object

Syntax

```
add(obj, 'type', 'name', ...)
add(obj, 'DriverSession', 'name', 'ModuleName', 'HardwareAssetName',
'P1', V1)
add(obj, 'HardwareAsset', 'name', 'IOResourceDescriptor', 'P1', V1)
add(obj, 'LogicalName', 'name', 'SessionName', 'P1', V1)
add(obj, struct)
```

Arguments

obj	IVI configuration store object
'DriverSession' 'HardwareAsset' 'LogicalName'	Type of entry being added
'name'	Name of the DriverSession, HardwareAsset, or LogicalName being added
'IOResourceDescriptor'	Tells the driver exactly how to locate the device this asset represents
'ModuleName'	IVI instrument driver or software module
'HardwareAssetName'	Unique identifier for hardware asset
'SessionName'	Unique identifier for asset driver session
'P1'	First optional parameter for added entry. Other parameter-value pairs may follow.
V1	Value for first parameter
struct	Structure defining entry to be added; field names are the entry parameter names

Description

add(obj, '*type*', 'name', ...) adds a new entry of *type* to the IVI configuration store object, obj, with name, name. If an entry of type, *type*, with name, name, already exists an error will occur. Based on *type*, additional arguments are required. *type* can be HardwareAsset, DriverSession, or LogicalName.

add(obj, '**DriverSession**', 'name', 'ModuleName', 'HardwareAssetName', '*P1*', V1) adds a new driver session entry to the IVI configuration store object, obj, with name, name, using the specified software module name, ModuleName and hardware asset name, HardwareAssetName. Optional parameter-value pairs may be included.

Valid parameters for DriverSession are listed below. The default value for on/off parameters is off.

Parameter	Value	Description
Description	Any string	Description of driver session
VirtualNames	structure	A struct array containing virtual name mappings
Cache	on/off	Enable caching if the driver supports it.
DriverSetup	Any string	This value is software module dependent
InterchangeCheck	on/off	Enable driver interchangeability checking, if supported
QueryInstrStatus	on/off	Enable instrument status querying by the driver
RangeCheck	on/off	Enable extended range checking by the driver, if supported
RecordCoercions	on/off	Enable recording of coercions by the driver, if supported
Simulate	on/off	Enable simulation by the driver

add(obj, '**HardwareAsset**', 'name', 'IOResourceDescriptor', '*P1*', V1) adds a new hardware asset entry to the IVI configuration store object, obj, with name, name, and resource descriptor, IOResourceDescriptor. Optional parameter-value pairs may be included.

Valid parameters for HardwareAsset are

Parameter	Value	Description
`Description`	Any string	Description of hardware asset

`add(obj, '`**`LogicalName`**`', 'name', 'SessionName', '`*`P1`*`', V1)` adds a new logical name entry to the IVI configuration store object, `obj`, with name, `name`, and driver session name, `SessionName`. Optional parameter-value pairs may be included.

Valid parameters for `LogicalName` are

Parameter	Value	Description
`Description`	Any string	Description of logical name

`add(obj, struct)`, where `struct` is a structure whose field names are the entry parameter names, adds an entry to the IVI configuration store object, `obj`, of the specified type with the values contained in the structure.

Additions made to the configuration store object, `obj`, can be saved to the configuration store data file with the `commit` function.

Examples

Construct IVI configuration store object, `c`.

```
c = iviconfigurationstore;
```

Add a hardware asset with name `gpib1`, and resource description `GPIBO::1::INSTR`.

```
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::ISNTR`.

```
s.Type = 'HardwareAsset';
s.Name = 'gpib3';
s.IOResourceDescriptor = 'GPIB0::3::INSTR';
add(c, s);
```

Save the changes to the IVI configuration store data file.

```
commit(c);
```

See Also
iviconfigurationstore | commit | remove | update

binblockread

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 the MATLAB values to the precision specified by *precision*. By default the `uchar` precision is used and numeric values are returned in double-precision arrays. Refer to the `fread` function for a list of supported precisions.

`[A,count] = binblockread(...)` returns the number of values read to `count`.

`[A,count,msg] = binblockread(...)` returns a warning message to `msg` if the read operation did not complete successfully.

Examples

Create the GPIB object `g` associated with a National Instruments GPIB controller with board index 0, and a Tektronix TDS 210 oscilloscope with primary address 2.

```
g = gpib('ni',0,2);
g.InputBufferSize = 3000;
```

Connect `g` to the instrument, and write string commands that configure the scope to transfer binary waveform data from memory location A.

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

Write the `CURVE?` command, which prepares the scope to transfer data, and read the data using the binblock format.

```
fprintf(g,'CURVE?')
data = binblockread(g);
```

More About

Tips

Before you can read data from the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

binblockread blocks the MATLAB Command Window until one of the following occurs:

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

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

Each time binblockread is issued, the `ValuesReceived` property value is increased by the number of values read.

Some instruments may send a terminating character after the binblock. binblockread will not read the terminating character. You can read the terminating character with the `fread` function. Additionally, if `obj` is a GPIB, VISA-GPIB, VISA-VXI, VISA-USB, or VISA-RSIB object, you can use the `clrdevice` function to remove the terminating character.

Note: If you do not set the terminator property to `''` (null) before you execute `fprintf` or `fwrite`, `binblockread` may return incomplete data.

See Also
`binblockwrite` | `fopen` | `fread` | `instrhelp` | `BytesAvailable` | `InputBufferSize` | `Status` | `ValuesReceived`

binblockwrite

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 the MATLAB values to the precision specified by *precision*. By default the uchar precision is used. Refer to the fwrite function for a list of supported precisions.

binblockwrite(obj,A,'header') writes a binblock using the data, A, and the ASCII header, header, to the instrument connected to interface object, obj. The data written is constructed using the formula

```
<header>#<N><D><A>
```

binblockwrite(obj,A,'*precision*','header') writes binary data, A, translating the MATLAB values to the specified precision, *precision*. The ASCII header, header, is prefixed to the binblock.

binblockwrite(obj,A,'*precision*','header','*headerformat*') writes binary data, A, translating the MATLAB values to the specified precision, *precision*. The ASCII header, header, is prefixed to the binblock using the format specified by *headerformat*.

headerformat is a string containing C language conversion specifications. Conversion specifications are composed of the character % and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Type instrhelp fprintf for more information on valid values for *headerformat*. By default, *headerformat* is %s.

Examples

```
s = visa('ni', 'ASRL2::INSTR');
fopen(s);

% Write the command: [double('#14') 0 5 0 5] to the instrument.
binblockwrite(s, [0 5 0 5]);

% Write the command: [double('Curve #14') 0 5 0 5] to the
% instrument.
binblockwrite(s, [0 5 0 5], 'Curve ')
fclose(s);
```

More About

Tips

Before you can write data to the instrument, it must be connected to obj with the fopen function. A connected interface object has a Status property value of open. An error is

returned if you attempt to perform a write operation while `obj` is not connected to the instrument.

The `ValuesSent` property value is increased by the number of values written each time `binblockwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

See Also

binblockread | fopen | fwrite | instrhelp | OutputBufferSize | OutputEmptyFcn | Status | Timeout | TransferStatus | ValuesSent

bluetooth

Create Bluetooth object

Syntax

```
B = Bluetooth('RemoteName', Channel)
B = Bluetooth('RemoteID', Channel)
B = Bluetooth('RemoteID', Channel, 'P1',V1,'P2',V2,...)
```

Description

The Instrument Control Toolbox Bluetooth interface lets you connect to devices over the Bluetooth interface and to transmit and receive ASCII and binary data. Instrument Control Toolbox supports the Bluetooth Serial Port Profile (SPP). You can identify any SPP Bluetooth device and establish a two-way connection with that device.

`B = Bluetooth('RemoteName', Channel)` constructs a Bluetooth object associated with the `RemoteName` and `Channel`. `RemoteName` is a friendly way to identify the `RemoteID`. If a channel is not specified, it will default to `0`.

`B = Bluetooth('RemoteID', Channel)` constructs a Bluetooth object directly from the `RemoteID` and `Channel`.

In order to communicate with the Bluetooth device, use the `fopen` function. When the Bluetooth object is constructed, the object's `status` property is `closed`. Once the object is connected to the remote device with the `fopen` function, the `status` property is configured to `open`.

`B = Bluetooth('RemoteID', Channel, 'P1',V1,'P2',V2,...)` constructs a Bluetooth object associated with the `RemoteID`, `Channel` and with the specified property values. If an invalid property name or property value is specified the object will not be created. The property value pairs can be in any format supported by the `set` function, i.e., param-value string pairs, structures, and param-value cell array pairs.

For information on other functions that can be used with `Bluetooth`, a full example using the Bluetooth interface, events and callbacks, and usage guidelines see "Bluetooth Interface Overview" on page 8-2.

Properties that can be used with the `Bluetooth` object include:

Property	Description
`Channel`	Use to specify a channel if the device has channels. If none is provided, it defaults to `0`.
`RemoteName`	"Friendly name" for the Bluetooth device. For example, in the case of an iPhone, it might be simply `'iPhone'` or a name like `'Zor'`.
	This property is a string and can be empty. If it is empty, you must use the `RemoteID` to communicate with the device.
`RemoteID`	Internal ID of the Bluetooth device, equivalent to the Device ID. This is usually a 12-digit string that starts with `btspp://`. For example, `'btspp://0016530FD65F'`
	This property is a string and every device has one. You can use this or the `RemoteName` to communicate with the device.

Examples

Find available Bluetooth devices.

```
instrhwinfo('Bluetooth');
instrhwinfo('Bluetooth', RemoteName);
```

Construct a Bluetooth object called `b` using channel `3` of a Lego Mindstorm robot with `RemoteName` of NXT.

```
b = Bluetooth('NXT', 3);
```

Connect to the remote device.

```
fopen(b)
```

Send a message to the remote device using the `fwrite` function.

```
fwrite(b, uint8([2,0,1,155]));
```

Read data from the remote device using the `fread` function.

```
name = fread(b,35);
```

Disconnect the Bluetooth device.

```
fclose(b);
```

Clean up by deleting and clearing the object.

```
fclose(b);
clear(b);
```

More About

- "Bluetooth Interface Overview" on page 8-2

clear

Remove instrument objects from MATLAB workspace

Syntax

```
clear obj
```

Arguments

obj   An instrument object or an array of instrument objects.

Description

clear obj removes obj from the MATLAB workspace.

Examples

This example creates the GPIB object g, copies g to a new variable gcopy, and clears g from the MATLAB workspace. g is then restored to the workspace with instrfind and is shown to be identical to gcopy.

```
g = gpib('ni',0,1);
gcopy = g;
clear g
g = instrfind;
isequal(gcopy,g)
ans =
     1
```

More About

Tips

If obj is connected to the instrument and it is cleared from the workspace, then obj remains connected to the instrument. You can restore obj to the workspace with the

`instrfind` function. An object connected to the instrument has a `Status` property value of `open`.

To disconnect `obj` from the instrument, use the `fclose` function. To remove `obj` from memory, use the `delete` function. You should remove invalid instrument objects from the workspace with `clear`.

See Also

delete | fclose | instrfind | instrhelp | isvalid | Status

clrdevice

Clear instrument buffer

Syntax

```
clrdevice(obj)
```

Arguments

obj	A GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-Serial, or VISA-TCPIP object.

Description

clrdevice(obj) clears the hardware buffer of the instrument connected to obj.

More About

Tips

Before you can clear the hardware buffer, the instrument must be connected to obj with the fopen function. A connected object has a Status property value of open. If you issue clrdevice when obj is disconnected from the instrument, then an error is returned.

You can clear the software input buffer using the flushinput function. You can clear the software output buffer using the flushoutput function.

See Also
flushinput | flushoutput | fopen | Status

commit

Save IVI configuration store object to data file

Syntax

```
commit(obj)
commit(obj, 'file')
```

Arguments

obj	IVI configuration store object
'file'	Configuration store data file

Description

commit(obj) saves the IVI configuration store object, obj, to the configuration store data file. The configuration store data file is defined by obj's ActualLocation property.

commit(obj, 'file') saves the IVI configuration store object, obj, to the configuration store data file, file. No changes are saved to the configuration store data file that is defined by obj's ActualLocation property.

The IVI configuration store object can be modified with the add, update, and remove functions.

See Also
iviconfigurationstore | add | remove | update

connect

Connect device object to instrument

Syntax

```
connect(obj)
connect(obj,'update')
```

Arguments

obj	A device object or an array of device objects.
update	Update the state of the object or the instrument.

Description

connect(obj) connects the device object specified by obj to the instrument. obj can be an array of device objects.

connect(obj,'update') updates the state of the object or the instrument. update can be object or instrument. If update is object, the object is updated to reflect the state of the instrument. If update is instrument, the instrument is updated to reflect the state of the object. In this case, all property values defined by the object are sent to the instrument on open. By default, update is object.

Examples

Create a device object for a Tektronix TDS 210 oscilloscope that is connected to a National Instruments GPIB board.

```
g = gpib('ni',0,2);
d = icdevice('tektronix_tds210',g);
```

Connect to the instrument.

```
connect(d)
```

List the oscilloscope settings that can be configured.

```
props = set(d);
```

Get the current configuration of the oscilloscope.

```
values = get(d);
```

Disconnect from the instrument and clean up.

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

More About

Tips

If `obj` is successfully connected to the instrument, its `Status` property is configured to `open`. If `obj` is an array of device objects and one of the objects cannot be connected to the instrument, the remaining objects in the array will be connected and a warning is displayed.

See Also
`disconnect` | `delete` | `instrhelp` | `Status`

delete

Remove instrument objects from memory

Syntax

```
delete(obj)
```

Arguments

obj	An instrument object or an array of instrument objects.

Description

delete(obj) removes obj from memory.

Examples

This example creates the GPIB object g, connects g to the instrument, writes and reads text data, disconnects g, removes g from memory using delete, and then removes g from the workspace using clear.

```
g = gpib('ni',0,1);
fopen(g)
fprintf(g,'*IDN?')
idn = fscanf(g);
fclose(g)
delete(g)
clear g
```

More About

Tips

When you delete obj, it becomes an *invalid* object. Because you cannot connect an invalid object to the instrument, you should remove it from the workspace with the

`clear` command. If multiple references to `obj` exist in the workspace, then deleting one reference invalidates the remaining references.

If `obj` is connected to the instrument, it has a `Status` property value of `open`. If you issue `delete` while `obj` is connected, the connection is automatically broken. You can also disconnect `obj` from the instrument with the `fclose` function.

If `obj` is an interface object that is associated with a device object, the device object is automatically deleted when `obj` is deleted. However, if `obj` is a device object, the interface object is not automatically deleted when `obj` is deleted.

See Also
`clear` | `fclose` | `instrhelp` | `isvalid` | `stopasync` | `Status`

devicereset

Reset instrument

Syntax

```
devicereset(obj)
```

Arguments

`obj`	A device object.

Description

`devicereset(obj)` resets the instrument associated with the device object specified by `obj`.

disconnect

Disconnect device object from instrument

Syntax

```
disconnect(obj)
```

Arguments

obj	A device object or an array of device objects.

Description

disconnect(obj) disconnects the device object specified by obj from the instrument.

Examples

Create a device object for a Tektronix TDS 210 oscilloscope that is connected to a National Instruments GPIB board.

```
g = gpib('ni',0,2);
d = icdevice('tektronix_tds210',g);
```

Connect to the instrument.

```
connect(d)
```

Get the current configuration of the oscilloscope.

```
values = get(d);
```

Disconnect from the instrument and clean up.

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

More About

Tips

If obj is disconnected from the instrument, its Status property is configured to closed. You can reconnect to the instrument with the connect function. If obj is an array of device objects and one of the objects cannot be disconnected from the instrument, the remaining objects in the array will be disconnected and a warning is displayed.

See Also
connect | delete | instrhelp | Status

disp

Display instrument object summary information

Syntax

```
obj
disp(obj)
```

Arguments

obj   An instrument object or an array of instrument objects.

Description

`obj` or `disp(obj)` displays summary information for `obj`.

Examples

The following commands display summary information for the GPIB object `g`.

```
g = gpib('ni',0,1)
g.EOSMode = 'read'
g
```

More About

Tips

In addition to the syntax shown above, you can display summary information for `obj` by excluding the semicolon when

· Creating an instrument object

- Configuring property values using the dot notation

You can also display summary information via the Workspace browser by right-clicking an instrument object, and selecting **Display Summary** from the context menu.

echotcpip

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 and Connect to an Echo Server

Shows how to set up an echo server.

Start the echo server on port 4000 and create a TCPIP object.

```
echotcpip('on',4000)
t = tcpip('localhost',4000);
```

Connect the TCPIP object to the host.

```
fopen(t)
```

Read and Write to the Echo Server

Shows how to communicate with the echo server.

Write to the host and read from the host.

```
fprintf(t,'echo this string.')
data = fscanf(t);
```

Display the read data.

```
data
```

The string is returned.

```
data =
echco this string.
```

Stop and Disconnect the Echo Server

Shows how to dismiss an echo server.

Stop the echo server and disconnect the TCPIP object from the host.

```
echotcpip('off')
fclose(t)
```

See Also
echoudp | udp | tcpip

echoudp

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
echotcpip | udp | tcpip

fclose

Disconnect interface object from instrument

Syntax

```
fclose(obj)
```

Arguments

obj	An interface object or an array of interface objects.

Description

`fclose(obj)` disconnects `obj` from the instrument.

Examples

This example creates the GPIB object g, connects g to the instrument, writes and reads text data, and then disconnects g from the instrument using `fclose`.

```
g = gpib('ni',0,1);
fopen(g)
fprintf(g,'*IDN?')
idn = fscanf(g);
fclose(g)
```

At this point, you can once again connect an interface object to the instrument. If you no longer need g, you should remove it from memory with the `delete` function, and remove it from the workspace with the `clear` command.

More About

Tips

If obj was successfully disconnected, then the Status property is configured to closed and the RecordStatus property is configured to off. You can reconnect obj to the instrument using the fopen function.

An error is returned if you issue fclose while data is being written asynchronously. In this case, you should abort the write operation with the stopasync function, or wait for the write operation to complete.

See Also

clear | delete | fopen | instrhelp | record | stopasync | RecordStatus | Status

fgen

Create Quick-Control Function Generator object

Syntax

```
myFGen = fgen()
connect(myFGen);
set(myFGen, 'P1',V1,'P2',V2,...)
enableOutput(myFGen);
```

Description

The Quick-Control Function Generator can be used for any function generator that uses an underlying IVI-C driver. However, you do not have to directly deal with the underlying driver. This fgen object is easy to use.

`myFGen = fgen()` creates an instance of the function generator named `myFGen`.

`connect(myFGen);` connects to the function generator.

`set(myFGen, 'P1',V1,'P2',V2,...)` assigns the specified property values.

`enableOutput(myFGen);` enables the function generator to produce a signal that appears at the output connector.

For information on the prerequisites for using fgen, see "Quick-Control Function Generator Prerequisites" on page 13-34.

The Quick-Control Function Generator `fgen` function can use the following special functions, in addition to standard functions such as `connect` and `disconnect`.

Function	Description
selectChannel	Specifies the channel name from which the function generator produces the waveform. Example: `selectChannel(myFGen, '1');`

Function	Description
`getDrivers`	Returns a list of available function generator instrument drivers. Example: `drivers = getDrivers(myFGen);`
`getResources`	Retrieves a list of available instrument resources. It returns a list of available VISA resource strings when using an IVI-C function generator. Example: `res = getResources(myFGen);`
`selectWaveform`	Specifies which arbitrary waveform the function generator produces. Example: `selectWaveform (myFGen, wh);` where wh is the waveform handle you are selecting.

`downloadWaveform`	Downloads an arbitrary waveform to the function generator. If you provide an output variable, a waveform handle is returned. It can be used in the `selectWaveform` and `removeWaveform` functions.
	
	If you don't provide an output variable, function generator will overwrite the waveform when a new waveform is downloaded and deletes it upon disconnection.
	
	Example:
	
	`% To download the following waveform to fgen`
	`w = 1:0.001:2;`
	`downloadWaveform (myFGen, w);`
	
	`% To download a waveform to fgen and return a`
	`    waveform handle`
	`wh = downloadWaveform (myFGen, w);`

Function	Description

removeWaveform	Removes a previously created arbitrary waveform from the function generator's memory. If a waveform handle is provided, it removes the waveform represented by the waveform handle.
	
	Example:
	
	``
	% Remove a waveform from fgen with waveform
	    handle 10000
	  removeWaveform (myFGen, 10000);
	``

enableOutput	Enables the function generator to produce a signal that appears at the output connector. This function produces a waveform defined by the Waveform property. If the Waveform property is set to 'Arb', the function uses the latest internal waveform handle to output the waveform. ``` enableOutput (myFGen); ```
disableOutput	Disables the signal that appears at the output connector. Disables the selected channel. ``` disableOutput (myFGen); ```
reset	Sets the function generator to factory state.

Arguments

The Quick-Control Function Generator `fgen` can use the following properties.

Property	Description
AMDepth	Specifies the extent of Amplitude modulation the function generator applies to the carrier signal. The units are a percentage of full modulation. At 0% depth, the output amplitude equals the carrier signal's amplitude. At 100% depth, the output amplitude equals twice the carrier signal's amplitude. This property affects function generator behavior only when the Mode is set to 'AM' and ModulationResource is set to 'internal'.
Amplitude	Specifies the amplitude of the standard waveform. The value is the amplitude at the output terminal. The units are volts peak-to-peak (Vpp). For example, to produce a waveform ranging from -5.0 to +5.0 volts, set this value to 10.0 volts. Does not apply if Waveform is of type 'Arb'.
ArbWaveformGain	Specifies the factor by which the function generator scales the arbitrary waveform data. Use this property to scale the arbitrary waveform to ranges other than -1.0 to +1.0. When set to 2.0, the output signal ranges from -2.0 to +2.0 volts. Only applies if Waveform is of type 'Arb'.
BurstCount	Specifies the number of waveform cycles that the function generator produces after it receives a trigger. Only applies if Mode is set to 'burst'.
ChannelNames	This read-only property provides available channel names in a cell array.
Driver	This property is optional. Use only if necessary to specify the underlying driver used to communicate with an instrument. If the DriverDetectionMode property is set to 'manual', use the Driver property to specify the instrument driver.
DriverDetectionMode	Sets up criteria for connection. Valid values are 'auto' and 'manual'. The default value is 'auto', which means you do not need to set a driver name

Property	Description
	before connecting to an instrument. If set to `'manual'`, a driver name needs to be provided using the `Driver` property before connecting to instrument.
`FMDeviation`	Specifies the maximum frequency deviation the modulating waveform applies to the carrier waveform. This deviation corresponds to the maximum amplitude level of the modulating signal. The units are Hertz (Hz). This property affects function generator behavior only when `Mode` is set to `'FM'` and `ModulationSource` is set to `'internal'`.
`Frequency`	Specifies the rate at which the function generator outputs an entire arbitrary waveform when Waveform is set to `'Arb'`. It specifies the frequency of the standard waveform when Waveform is set to standard waveform types. The units are Hertz (Hz).
`Mode`	Specifies run mode. Valid values are `'continuous'`, `'burst'`, `'AM'`, or `'FM'`. Specifies how the function generator produces waveforms. It configures the instrument to generate output continuously or to generate a discrete number of waveform cycles based on a trigger event. It can also be set to AM and FM.
`ModulationFrequency`	Specifies the frequency of the standard waveform that the function generator uses to modulate the output signal. The units are Hertz (Hz). This attribute affects function generator behavior only when `Mode` is set to `'AM'` or `'FM'` and the `ModulationSource` attribute is set to `'internal'`.
`ModulationSource`	Specifies the signal that the function generator uses to modulate the output signal. Valid values are `'internal'` and `'external'`. This attribute affects function generator behavior only when `Mode` is set to `'AM'` or `'FM'`.

Property	Description
ModulationWaveform	Specifies the standard waveform type that the function generator uses to modulate the output signal. This affects function generator behavior only when Mode is set to 'AM' or 'FM' and the ModulationSource is set to 'internal'. Valid values are 'sine', 'square', 'triangle', 'RampUp', 'RampDown', and 'DC'.
Offset	Uses the standard waveform DC offset as input arguments if the waveform is not of type 'Arb'. Use Arb Waveform Offset as input arguments if the waveform is of type 'Arb'. Specifies the DC offset of the standard waveform when Waveform is set to standard waveform. For example, a standard waveform ranging from +5.0 volts to 0.0 volts has a DC offset of 2.5 volts. When Waveform is set to 'Arb', this property shifts the arbitrary waveform's range. For example, when it is set to 1.0, the output signal ranges from 2.0 volts to 0.0 volts.
OutputImpedance	Specifies the function generator's output impedance at the output connector.
Resource	Set this before connecting to the instrument. It is the VISA resource string for your instrument.
SelectedChannel	Returns the selected channel name that was set using the selectChannel function.
StartPhase	Specifies the horizontal offset in degrees of the standard waveform the function generator produces. The units are degrees of one waveform cycle. For example, a 180-degree phase offset means output generation begins halfway through the waveform.
Status	This read-only property indicates the communication status of your instrument session. It is either 'open' or 'closed'.

Property	Description
`TriggerRate`	Specifies the rate at which the function generator's internal trigger source produces a trigger, in triggers per second. This property affects function generator behavior only when the `TriggerSource` is set to `'internal'`. Only applies if `Mode` is set to `'burst'`.
`TriggerSource`	Specifies the trigger source. After the function generator receives a trigger, it generates an output signal if `Mode` is set to `'burst'`. Valid values are `'internal'` or `'external'`.
`Waveform`	Uses the waveform type as an input argument. Valid values are `'Arb'`, for an arbitrary waveform, or these standard waveform types – `'Sine'`, `'Square'`, `'Triangle'`, `'RampUp'`, `'RampDown'`, and `'DC'`.

Examples

Create an instance of the function generator called `myFGen`.

```
myFGen = fgen()
```

Discover available resources. A resource string is an identifier to the instrument. You need to set it before connecting to the instrument.

```
availableResources = getResources(myFGen)
```

Set the resource. In this example, we are controlling an instrument that is connected via GPIB with a board index of 0 and primary address of 10.

```
myFGen.Resource = 'GPIB0::10::INSTR';
```

Connect to the function generator.

```
connect(myFGen);
```

Specify the channel name from which the function generator produces the waveform.

```
selectChannel(myFGen, '1');
```

Configure a standard waveform to be a continuous sine wave.

```
set(myFGen, 'Waveform', 'sine');
set(myFGen, 'Mode', 'continuous');
```

Configure the function generator.

```
% Set the load impedance to 50 Ohms.
set(myFGen, 'OutputImpedance', 50);

% Set the frequency to 2500 Hz.
set(myFGen, 'Frequency', 2500);

% Set the amplitude to 1.2 volts.
set(myFGen, 'Amplitude', 1.2);

% Set the offset to 0.4 volts.
set(myFGen, 'Offset', 0.4);
```

Communicate with the instrument. For example, output signals. In this example, the `enableOutput` function enables the function generator to produce a signal that appears at the output connector.

```
% Enable the output of signals.
```

```
enableOutput(myFGen);
```

When you are done, disable the output.

```
% Disable the output of signals.
disableOutput(myFGen);
```

Close the session and remove it from the workspace.

```
disconnect(myFGen);
delete myFGen;
clear myFGen;
```

These examples used a standard waveform type. For examples using an arbitrary waveform, see "Generating Waveforms Using the Quick-Control Function Generator" on page 13-35.

More About

· "Using Quick-Control Function Generator" on page 13-34

fgetl

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.

22-45

Examples

Create the GPIB object g, connect g to a Tektronix TDS 210 oscilloscope, configure g to complete read operations when the End-Of-String character is read, and write the *IDN? command with the fprintf function. *IDN? instructs the scope to return identification information.

```
g = gpib('ni',0,1);
fopen(g)
g.EOSMode = 'read';
fprintf(g,'*IDN?')
```

Asynchronously read the identification information from the instrument.

```
readasync(g)
g.BytesAvailable
ans =
    56
```

Use fgetl to transfer the data from the input buffer to the MATLAB workspace, and discard the terminator.

```
idn = fgetl(g)
idn =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04

length(idn)
ans =
    55
```

Disconnect g from the scope, and remove g from memory and the workspace.

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

More About

Tips

Before you can read text from the instrument, it must be connected to obj with the fopen function. A connected interface object has a Status property value of open. An

error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read — including the terminator — each time `fgetl` is issued.

Note: You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

Rules for Completing a Read Operation with fgetl

A read operation with `fgetl` blocks access to the MATLAB Command Window until

- The terminator is read. For serial port, TCPIP, UDP, and VISA-serial objects, the terminator is given by the `Terminator` property. Note that for UDP objects, `DatagramTerminateMode` must be `off`.

  For all other interface objects except VISA-RSIB, the terminator is given by the `EOSCharCode` property.
- The EOI line is asserted (GPIB and VXI instruments only).
- A datagram has been received (UDP objects only if `DatagramTerminateMode` is `on`).
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Note: You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

More About the GPIB and VXI Terminator

The `EOSCharCode` property value is recognized only when the `EOSMode` property is configured to `read` or `read&write`. For example, if `EOSMode` is configured to `read` and `EOSCharCode` is configured to `LF`, then one of the ways that the read operation terminates is when the line feed character is received.

If EOSMode is none or write, then there is no terminator defined for read operations. In this case, fgetl will complete execution and return control to the command line when another criterion, such as a timeout, is met.

See Also

fgets | fopen | instrhelp | BytesAvailable | EOSCharCode | EOSMode | InputBufferSize | Status | Terminator | Timeout | ValuesReceived

fgets

Read line of text from instrument and include terminator

Syntax

```
tline = fgets(obj)
[tline,count] = fgets(obj)
[tline,count,msg] = fgets(obj)
[tline,count,msg,datagramaddress,datagramport] = fgets(obj)
```

Arguments

obj	An interface object.
tline	The text read from the instrument, including the terminator.
count	The number of values read.
msg	A message indicating that the read operation did not complete successfully.
datagramaddress	The datagram address.
datagramport	The datagram port.

Description

`tline = fgets(obj)` reads one line of text from the instrument connected to `obj`, and returns the data to `tline`. The returned data includes the terminator with the text line. To exclude the terminator, use `fgetl`.

`[tline,count] = fgets(obj)` returns the number of values read to `count`.

`[tline,count,msg] = fgets(obj)` returns a warning message to `msg` if the read operation was unsuccessful.

`[tline,count,msg,datagramaddress,datagramport] = fgets(obj)` returns the remote address and port from which the datagram originated. These values are returned only if `obj` is a UDP object.

Examples

Create the GPIB object g, connect g to a Tektronix TDS 210 oscilloscope, configure g to complete read operations when the End-Of-String character is read, and write the `*IDN?` command with the `fprintf` function. `*IDN?` instructs the scope to return identification information.

```
g = gpib('ni',0,1);
fopen(g)
g.EOSMode = 'read';
fprintf(g,'*IDN?')
```

Asynchronously read the identification information from the instrument.

```
readasync(g)
g.BytesAvailable
ans =
    56
```

Use `fgets` to transfer the data from the input buffer to the MATLAB workspace, and include the terminator.

```
idn = fgets(g)
idn =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
length(idn)
ans =
    56
```

Disconnect g from the scope, and remove g from memory and the workspace.

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

More About

Tips

Before you can read text from the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An

error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read — including the terminator — each time `fgets` is issued.

Note: You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

Rules for Completing a Read Operation with fgets

A read operation with `fgets` blocks access to the MATLAB command line until

- The terminator is read. For serial port, TCPIP, UDP, and VISA-serial objects, the terminator is given by the `Terminator` property. Note that for UDP objects, `DatagramTerminateMode` must be `off`.

  For all other interface objects except VISA-RSIB, the terminator is given by the `EOSCharCode` property.
- The EOI line is asserted (GPIB and VXI instruments only).
- A datagram has been received (UDP objects only if `DatagramTerminateMode` is `on`).
- The time specified by the `Timeout` property passes.
- The input buffer is filled.

Note: You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

More About the GPIB and VXI Terminator

The `EOSCharCode` property value is recognized only when the `EOSMode` property is configured to `read` or `read&write`. For example, if `EOSMode` is configured to `read` and `EOSCharCode` is configured to `LF`, then one of the ways that the read operation terminates is when the line feed character is received.

If `EOSMode` is `none` or `write`, then there is no terminator defined for read operations. In this case, `fgets` will complete execution and return control to the command line when another criterion, such as a timeout, is met.

See Also

fgetl | fopen | instrhelp | query | BytesAvailable | EOSCharCode | EOSMode | InputBufferSize | Status | Terminator | Timeout | ValuesReceived

flushinput

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.

More About

Tips

After the input buffer is flushed, the BytesAvailable property is automatically configured to 0.

If flushinput is called during an asynchronous (nonblocking) read operation, the data currently stored in the input buffer is flushed and the read operation continues. You can read data asynchronously from the instrument using the readasync function.

The input buffer is automatically flushed when you connect an object to the instrument with the fopen function.

You can clear the output buffer with the flushoutput function. You can clear the hardware buffer for GPIB and VXI instruments with the clrdevice function.

See Also
clrdevice | flushoutput | fopen | readasync | BytesAvailable

flushoutput

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`.

More About

Tips

After the output buffer is flushed, the `BytesToOutput` property is automatically configured to `0`.

If `flushoutput` is called during an asynchronous (nonblocking) write operation, the data currently stored in the output buffer is flushed and the write operation is aborted. Additionally, the callback function specified for the `OutputEmptyFcn` property is executed. You can write data asynchronously to the instrument using the `fprintf` or `fwrite` functions.

The output buffer is automatically flushed when you connect an object to the instrument with the `fopen` function.

You can clear the input buffer with the `flushinput` function. You can clear the hardware buffer for GPIB and VXI instruments with the `clrdevice` function.

See Also
clrdevice | flushinput | fopen | fprintf | fwrite | BytesToOutput |
OutputEmptyFcn

fopen

Connect interface object to instrument

Syntax

```
fopen(obj)
```

Arguments

obj   An interface object or an array of interface objects.

Description

fopen(obj) connects obj to the instrument.

Examples

This example creates the GPIB object g, connects g to the instrument using fopen, writes and reads text data, and then disconnects g from the instrument.

```
g = gpib('ni',0,1);
fopen(g)
fprintf(g,'*IDN?')
idn = fscanf(g);
fclose(g)
```

More About

Tips

Before you can perform a read or write operation, obj must be connected to the instrument with the fopen function. When obj is connected to the instrument

- Data remaining in the input buffer or the output buffer is flushed.
- The `Status` property is set to `open`.
- The `BytesAvailable`, `ValuesReceived`, `ValuesSent`, and `BytesToOutput` properties are set to 0.

An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the instrument. You can connect only one interface object to a given instrument. For example, on a Windows machine you can connect only one serial port object to an instrument associated with the COM1 port. Similarly, you can connect only one GPIB object to an instrument with a given board index, primary address, and secondary address.

Some properties are read-only while the interface object is connected, and must be configured before using `fopen`. Examples include `InputBufferSize` and `OutputBufferSize`. Refer to the property reference pages or use the `propinfo` function to determine which properties have this constraint.

The values for some properties are verified only after `obj` is connected to the instrument. If any of these properties are incorrectly configured, an error is returned when `fopen` is issued and `obj` is not connected to the instrument. Properties of this type include `BaudRate` and `SecondaryAddress`, and are associated with instrument settings.

See Also

`fclose` | `instrhelp` | `propinfo` | `BytesAvailable` | `BytesToOutput` | `Status` | `ValuesReceived` | `ValuesSent`

fprintf

Write text to instrument

Syntax

```
fprintf(obj,'cmd')
fprintf(obj,'format','cmd')
fprintf(obj,'cmd','mode')
fprintf(obj,'format','cmd','mode')
```

Arguments

`obj`	An interface object.
`'cmd'`	The string written to the instrument.
`'format'`	C language conversion specification.
`'mode'`	Specifies whether data is written synchronously or asynchronously.

Description

`fprintf(obj,'cmd')` writes the string `cmd` to the instrument connected to `obj`. The default format is `%s\n`. The write operation is synchronous and blocks the command line until execution is complete.

`fprintf(obj,'format','cmd')` writes the string using the format specified by *format*.

format is a C language conversion specification. Conversion specifications involve the `%` character and the conversion characters d, i, o, u, x, X, f, e, E, g, G, c, and s. Refer to the `sprintf` file I/O format specifications or a C manual for more information.

`fprintf(obj,'cmd','mode')` writes the string with command-line access specified by *mode*. If *mode* is `sync`, `cmd` is written synchronously and the command line is blocked.

If *mode* is `async`, `cmd` is written asynchronously and the command line is not blocked. If *mode* is not specified, the write operation is synchronous.

`fprintf(obj,'format','cmd','mode')` writes the string using the specified format. If *mode* is `sync`, `cmd` is written synchronously. If *mode* is `async`, `cmd` is written asynchronously.

Examples

Create the serial port object `s`, connect `s` on a Windows machine to a Tektronix TDS 210 oscilloscope, and write the `RS232?` command with the `fprintf` function. `RS232?` instructs the scope to return serial port communications settings.

```
s = serial('COM1');
fopen(s)
fprintf(s,'RS232?')
settings = fscanf(s)
settings =
9600;1;0;NONE;LF
```

Because the default format for `fprintf` is `%s\n`, the terminator specified by the `Terminator` property was automatically written. However, in some cases you might want to suppress writing the terminator. To do so, you must explicitly specify a format for the data that does not include the terminator, or configure the terminator to empty.

```
fprintf(s,'%s','RS232?')
```

More About

Tips

Before you can write text to the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the instrument.

The `ValuesSent` property value is increased by the number of values written each time `fprintf` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

`fprintf` function will return an error message if you set the `flowcontrol` property to `hardware` on a serial object, and a hardware connection is not detected. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Note: If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is `off`, there is a problem on the remote device side. If `ClearToSend` is `on`, there is a hardware `FlowControl` device prepared to receive data and you can execute `fprintf`.

Synchronous Versus Asynchronous Write Operations

By default, text is written to the instrument synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *mode* input argument to be `async`. For asynchronous writes,

- The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.
- The callback function specified for the `OutputEmptyFcn` property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the `TransferStatus` property.

Synchronous and asynchronous write operations are discussed in more detail in "Synchronous Versus Asynchronous Write Operations" on page 3-16.

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.

See Also
`fopen` | `fwrite` | `instrhelp` | `query` | `sprintf` | `BytesToOutput` | `EOSCharCode` | `EOSMode` | `OutputBufferSize` | `OutputEmptyFcn` | `Status` | `TransferStatus` | `ValuesSent`

fread

Read binary data from instrument

Syntax

```
A = fread(obj)
A = fread(obj,size)
A = fread(obj,size,'precision')
[A,count] = fread(...)
[A,count,msg] = fread(...)
[A,count,msg,datagramaddress] = fread(obj,...)
[A,count,msg,datagramaddress,datagramport] = fread(obj,...)
```

Arguments

`obj`	An interface object.
`size`	The number of values to read.
`'precision'`	The number of bits read for each value, and the interpretation of the bits as character, integer, or floating-point values.
`A`	Binary data returned from the instrument.
`count`	The number of values read.
`msg`	A message indicating if the read operation was unsuccessful.
`datagramaddress`	The address of the datagram sender.
`datagramport`	The port of the datagram sender.

Description

`A = fread(obj)` and `A = fread(obj,size)` read binary data from the instrument connected to `obj`, and returns the data to `A`. The maximum number of values to read is specified by `size`. If `size` is not specified, the maximum number of values to read is determined by the object's `InputBufferSize` property. Valid options for `size` are:

`n`	Read at most `n` values into a column vector.

[m,n]	Read at most m–by–n values filling an m–by–n matrix in column order.

`size` cannot be `inf`, and an error is returned if the specified number of values cannot be stored in the input buffer. You specify the size, in bytes, of the input buffer with the `InputBufferSize` property. A value is defined as a byte multiplied by the *precision* (see below).

If `obj` is a UDP object and `DatagramTerminateMode` is `off`, the `size` value is honored. If `size` is less than the length of the datagram, only `size` values are read. If `size` is greater than the length of the datagram, a warning is issued stating that a complete datagram was read before `size` values was reached.

`A = fread(obj,size,'`*precision*`')` reads binary data with precision specified by *precision*.

precision controls the number of bits read for each value and the interpretation of those bits as integer, floating-point, or character values. If *precision* is not specified, `uchar` (an 8-bit unsigned character) is used. By default, numeric values are returned in double-precision arrays. The supported values for *precision* are listed below in Tips.

`[A,count] = fread(...)` returns the number of values read to `count`.

`[A,count,msg] = fread(...)` returns a warning message to `msg` if the read operation was unsuccessful.

`[A,count,msg,datagramaddress] = fread(obj,...)` returns the datagram address to `datagramaddress` if `obj` is a UDP object. If more than one datagram is read, `datagramaddress` is `''`.

`[A,count,msg,datagramaddress,datagramport] = fread(obj,...)` returns the datagram port to `datagramport` if `obj` is a UDP object. If more than one datagram is read, `datagramport` is `[]`.

More About

Tips

Before you can read data from the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An

error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read, each time `fread` is issued.

Rules for Completing a Binary Read Operation

A read operation with `fread` blocks access to the MATLAB Command Window until

- The specified number of values is read. For UDP objects, `DatagramTerminateMode` must be `off`.
- The time specified by the `Timeout` property passes.
- A datagram is received (for UDP objects only when `DatagramTerminateMode` is `on`).
- The input buffer is filled.
- The EOI line is asserted (GPIB and VXI instruments only).
- The `EOSCharCode` is received (GPIB and VXI instruments only).

Note: Set the terminator property to `''` (null), if appropriate, to ensure efficient throughput of binary data.

More About the GPIB and VXI Terminator

The `EOSCharCode` property value is recognized only when the `EOSMode` property is configured to `read` or `read&write`. For example, if `EOSMode` is configured to `read` and `EOSCharCode` is configured to `LF`, then one of the ways that the read operation terminates is when the line feed character is received.

If `EOSMode` is `none` or `write`, then there is no terminator defined for read operations. In this case, `fread` will complete execution and return control to the command when another criterion, such as a timeout, is met.

Supported Precisions

The supported values for *precision* are listed below.

Data Type	Precision	Interpretation
Character	uchar	8-bit unsigned character
	schar	8-bit signed character
	char	8-bit signed or unsigned character
Integer	int8	8-bit integer
	int16	16-bit integer
	int32	32-bit integer
	uint8	8-bit unsigned integer
	uint16	16-bit unsigned integer
	uint32	32-bit unsigned integer
	short	16-bit integer
	int	32-bit integer
	long	32- or 64-bit integer
	ushort	16-bit unsigned integer
	uint	32-bit unsigned integer
	ulong	32- or 64-bit unsigned integer
Floating-point	single	32-bit floating point
	float32	32-bit floating point
	float	32-bit floating point
	double	64-bit floating point
	float64	64-bit floating point

See Also

fgetl | fgets | fopen | fscanf | instrhelp | BytesAvailable |
InputBufferSize | Status | ValuesReceived

fscanf

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 [].

Examples

Create the serial port object s on a Windows machine and connect s to a Tektronix TDS 210 oscilloscope, which is displaying a sine wave.

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

Use the `fprintf` function to configure the scope to measure the peak-to-peak voltage of the sine wave, return the measurement type, and return the peak-to-peak voltage.

```
fprintf(s,'MEASUREMENT:IMMED:TYPE PK2PK')
fprintf(s,'MEASUREMENT:IMMED:TYPE?')
fprintf(s,'MEASUREMENT:IMMED:VAL?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data associated with the two query commands is automatically returned to the input buffer.

```
s.BytesAvailable
ans =
    13
```

Use `fscanf` to read the measurement type. The operation will complete when the first terminator is read.

```
meas = fscanf(s)
meas =
PK2PK
```

Use `fscanf` to read the peak-to-peak voltage as a floating-point number, and exclude the terminator.

```
pk2pk = fscanf(s,'%e',6)
pk2pk =
    2.0200
```

Disconnect `s` from the scope, and remove `s` from memory and the workspace.

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

More About

Tips

Before you can read data from the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read — including the terminator — each time `fscanf` is issued.

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.

See Also
`fgetl` | `fgets` | `fopen` | `fread` | `instrhelp` | `scanstr` | `sscanf` |
`BytesAvailable` | `BytesAvailableFcn` | `EOSCharCode` | `EOSMode` |
`InputBufferSize` | `Status` | `Terminator` | `Timeout` | `TransferStatus`

fwrite

Write binary data to instrument

Syntax

```
fwrite(obj,A)
fwrite(obj,A,'precision')
fwrite(obj,A,'mode')
fwrite(obj,A,'precision','mode')
```

Arguments

`obj`	An interface object.
`A`	The binary data written to the instrument.
`'precision'`	The number of bits written for each value, and the interpretation of the bits as character, integer, or floating-point values.
`'mode'`	Specifies whether data is written synchronously or asynchronously.

Description

`fwrite(obj,A)` writes the binary data A to the instrument connected to `obj`.

`fwrite(obj,A,'precision')` writes binary data with precision specified by *precision*.

precision controls the number of bits written for each value and the interpretation of those bits as integer, floating-point, or character values. If *precision* is not specified, `uchar` (an 8-bit unsigned character) is used. The support values for *precision* are listed in "Supported Precisions" on page 22-64.

`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*.

More About

Tips

Before you can write data to the instrument, it must be connected to `obj` with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a write operation while `obj` is not connected to the instrument.

The `ValuesSent` property value is increased by the number of values written each time `fwrite` is issued.

An error occurs if the output buffer cannot hold all the data to be written. You can specify the size of the output buffer with the `OutputBufferSize` property.

`fwrite` will return an error message if you set the `FlowControl` property to `hardware` on a serial object, and a hardware connection is not detected. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Note: If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is `off`, there is a problem on the remote device side. If `ClearToSend` is `on`, there is a hardware `FlowControl` device prepared to receive data and you can execute `fwrite`.

Synchronous Versus Asynchronous Write Operations

By default, data is written to the instrument synchronously and the command line is blocked until the operation completes. You can perform an asynchronous write by configuring the *mode* input argument to be `async`. For asynchronous writes,

• The `BytesToOutput` property value is continuously updated to reflect the number of bytes in the output buffer.

- The callback function specified for the OutputEmptyFcn property is executed when the output buffer is empty.

You can determine whether an asynchronous write operation is in progress with the TransferStatus property.

Synchronous and asynchronous write operations are discussed in more detail in "Synchronous Versus Asynchronous Write Operations" on page 3-16.

Rules for Completing a Write Operation with fwrite

A binary write operation using fwrite completes when

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

Note The Terminator and EOSCharCode properties are not used with binary write operations.

Supported Precisions

The supported values for *precision* are listed below.

Data Type	Precision	Interpretation
Character	uchar	8-bit unsigned character
	schar	8-bit signed character
	char	8-bit signed or unsigned character
Integer	int8	8-bit integer
	int16	16-bit integer
	int32	32-bit integer
	uint8	8-bit unsigned integer
	uint16	16-bit unsigned integer
	uint32	32-bit unsigned integer

Data Type	Precision	Interpretation
	short	16-bit integer
	int	32-bit integer
	long	32- or 64-bit integer
	ushort	16-bit unsigned integer
	uint	32-bit unsigned integer
	ulong	32- or 64-bit unsigned integer
Floating-point	single	32-bit floating point
	float32	32-bit floating point
	float	32-bit floating point
	double	64-bit floating point
	float64	64-bit floating point

See Also

fopen | fprintf | instrhelp | OutputBufferSize | OutputEmptyFcn | Status | Timeout | TransferStatus | ValuesSent

get

Instrument object properties

Syntax

```
get(obj)
out = get(obj)
out = get(obj,'PropertyName')
```

Arguments

`obj`	An instrument object or an array of instrument objects.
`'PropertyName'`	A property name or a cell array of property names.
`out`	A single property value, a structure of property values, or a cell array of property values.

Description

`get(obj)` returns all property names and their current values to the command line for `obj`. The properties are divided into two sections. The base properties are listed first and the object-specific properties are listed second.

`out = get(obj)` returns the structure `out` where each field name is the name of a property of `obj`, and each field contains the value of that property.

`out = get(obj,'PropertyName')` returns the value `out` of the property specified by *PropertyName* for `obj`. If *PropertyName* is replaced by a 1-by-n or n-by-1 cell array of 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.

Examples

This example illustrates some of the ways you can use `get` to return property values for the GPIB object `g`.

```
g = gpib('ni',0,1);
out1 = get(g);
out2 = get(g,{'PrimaryAddress','EOSCharCode'});
get(g,'EOIMode')
ans =
on
```

More About

Tips

When specifying a property name, you can do so without regard to case, and you can make use of property name completion. For example, if `g` is a GPIB object, then these commands are all valid.

```
out = get(g,'EOSMode');
out = get(g,'eosmode');
out = get(g,'EOSM');
```

See Also
instrhelp | set | propinfo

geterror

Check and return error message from instrument

Syntax

```
msg = geterror(obj)
```

Arguments

obj	A device object.
msg	The error message returned from the instrument.

Description

`msg = geterror(obj)` checks the instrument associated with the device object specified by `obj` for an error message. If an error message exists, it is returned to `msg`. The interpretation of `msg` will vary based on the instrument.

getWaveform

Returns waveform displayed on scope

Syntax

```
w = getWaveform(myScope);
w = getWaveform(myScope, 'acquisition', true);
w = getWaveform(myScope, 'acquisition', false);
```

Description

`w = getWaveform(myScope);` returns waveform(s) displayed on the scope screen. Retrieves the waveform(s) from enabled channel(s). By default it downloads the captured waveform from the scope without acquisition.

`w = getWaveform(myScope, 'acquisition', true);` initiates an acquisition and returns waveform(s) from the oscilloscope.

`w = getWaveform(myScope, 'acquisition', false);` gets waveform from the enabled channel without acquisition

This function can only be used with the `oscilloscope` object. You can use the `getWaveform` function to download the current waveform from the scope or to initiate the waveform and capture it. See the examples below for the three possible use cases.

Examples

Use this example if you have captured the waveform(s) using the oscilloscope's front panel and want to download it to the Instrument Control Toolbox for further analysis.

```
o = oscilloscope()
set (o, 'Resource', 'instrumentResourceString');
connect(o);
w = getWaveform(o);
```

Replace `'instrumentResourceString'` with the resource string for your instrument.

Use this example to get the waveform from a circuit output (without configuring the trigger) and download it to the Instrument Control Toolbox to check it.

```
o = oscilloscope()
set (o, 'Resource', 'instrumentResourceString');
connect(o);
enableChannel(o,'Channel1');
w = getWaveform(o);
```

Replace `'instrumentResourceString'` with the resource string for your instrument.

Use this example to capture synchronized input/output signals of a filter circuit when a certain trigger condition is met, stop the acquisition, and download the waveforms to the Instrument Control Toolbox.

```
o = oscilloscope()
set (o, 'Resource', 'instrumentResourceString');
connect(o);
set (o, 'TriggerMode','normal');
set (o, 'enableChannel', {'Channel1','Channel2'});
[w1, w2] = getWaveform(o, 'acqusition', true);
```

Replace `'instrumentResourceString'` with the resource string for your instrument.

More About

- "Using Quick-Control Oscilloscope" on page 13-26

gpib

Create GPIB object

Syntax

```
obj = gpib('vendor',boardindex,primaryaddress)
obj =
gpib('vendor',boardindex,primaryaddress,'PropertyName',PropertyValue)
```

Arguments

'*vendor*'	The vendor name.
boardindex	The GPIB board index.
primaryaddress	The instrument primary address.
'*PropertyName*'	A GPIB property name.
'PropertyValue'	A property value supported by *PropertyName*.
obj	The GPIB object.

Description

obj = gpib('*vendor*',boardindex,primaryaddress) creates the GPIB object obj associated with the board specified by boardindex, and the instrument specified by primaryaddress. The GPIB hardware is supplied by *vendor*. Supported vendors are given below.

Vendor	Description
advantech	Advantech Company hardware
agilent	Agilent Technologies hardware
cec	Capital Equipment Corporation hardware
contec	CONTEC hardware
ics	ICS Electronics hardware

Vendor	Description
iotech	IOTech hardware
keithley	Keithley Instruments hardware
mcc	Measurement Computing Corporation hardware
ni	National Instruments hardware

```
obj =
gpib('vendor',boardindex,primaryaddress,'PropertyName',PropertyValue)
```
creates the GPIB object with the specified property names and property values. If an invalid property name or property value is specified, an error is returned and `obj` is not created.

Examples

This example creates the GPIB object `g1` associated with a National Instruments board at index 0 with primary address 1, and then connects `g1` to the instrument.

```
g1 = gpib('ni',0,1);
fopen(g1)
```

The `Type`, `Name`, `BoardIndex`, and `PrimaryAddress` properties are automatically configured.

```
g1.Type
ans =
    gpib

g1.Name
ans =
    GPIB0-1

g1.BoardIndex
ans =
    0

g1.PrimaryAddress
ans =
    1
```

To specify the secondary address during object creation,

```
g2 = gpib('ni',0,1,'SecondaryAddress',96);
```

More About

Tips

At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with GPIB objects.

```
instrhelp gpib
```

When you create a GPIB object, these property value are automatically configured:

- `Type` is given by `gpib`.
- `Name` is given by concatenating `GPIB` with the board index and the primary address specified in the `gpib` function. If the secondary address is specified, then this value is also used in `Name`.
- `BoardIndex` and `PrimaryAddress` are given by the values supplied to the `gpib` function.

Note You do not use the GPIB board primary address in the GPIB object constructor syntax. You use the board index, and the instrument address.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, these commands are all valid:

```
g = gpib('ni',0,1,'SecondaryAddress',96);
g = gpib('ni',0,1,'secondaryaddress',96);
g = gpib('ni',0,1,'SECOND',96);
```

Before you can communicate with the instrument, it must be connected to `obj` with the `fopen` function. A connected GPIB object has a `Status` property value of `open`. An error is returned if you attempt to perform a read or write operation while `obj` is not connected to the instrument.

You cannot connect multiple GPIB objects to the same instrument. A GPIB instrument is uniquely identified by its board index, primary address, and secondary address.

See Also
fopen | instrhelp | instrhwinfo | BoardIndex | Name | PrimaryAddress |
SecondaryAddress | Status | Type

i2c

Create I2C object

Syntax

```
I = i2c('Vendor', BoardIndex, RemoteAddress)
```

Description

I2C, or Inter-Integrated Circuit, is a chip-to-chip interface supporting two-wire communication. Instrument Control Toolbox I2C support lets you open connections with individual chips and to read and write over the connections to individual chips using either an Aardvark host adaptor or a NI-845x adaptor board.

`I = i2c('Vendor', BoardIndex, RemoteAddress)` constructs an `i2c` object associated with `Vendor`, `BoardIndex`, and `RemoteAddress`. `Vendor` must be set to either `'aardvark'`, for use with a Total Phase Aardvark adaptor, or to `'NI845x'`, for use with a NI-845x adaptor board, to use this interface. `BoardIndex` specifies the board index of the hardware and is usually `0`. `RemoteAddress` specifies the remote address of the hardware. Note that to specify the remote address of 50 hex, you need to use the `hex2dec` function as shown in Examples.

The primary use cases involve the `fread` and `fwrite` functions. To identify I2C devices in the Instrument Control Toolbox, use the `instrhwinfo` function on the I2C interface, called `i2c`.

You can use these properties with the `i2c` object:

Property	Description
BitRate	Must be a positive, nonzero value specified in kHz. The adaptor and chips determine the rate. The default is 100 kHz for both the Aardvark and NI-845x adaptors.
TargetPower	Aardvark only. Can be specified as none or both. The value both means to power both lines, if supported. The value none means power no lines, and is the default value.

Property	Description
PullupResistors	Can be specified as none or both. The value both enables 2k pullup resistors to protect hardware in the I2C device, if supported. This is the default value.
	Note that devices may differ in their use of pullups. The Aardvark adaptor and the NI-8452 have internal pullup resistors to tie both bus lines to VDD and can be programmatically set. The NI-8451 does not have internal pullup resistors that can be programmatically set, and so require external pullups. You should consult your device documentation to ensure that the correct pullups have been used.
BoardSerial	Unique identifier of the I2C master communication device.
Vendor	Use to create i2c object. Must be set to aardvark, for use with Aardvark adaptor, or NI845x for use with the NI-845x adaptor.
BoardIndex	Use to create i2c object. Specifies the board index of the hardware. Usually set to 0.
RemoteAddress	Use to create i2c object. Specifies the remote address of the hardware. Specified as a string when you create the i2c object. For example, to specify the remote address of 50 hex, use '50h'.

Examples

Aardvark Example

This example shows how to communicate with an EEPROM chip on a circuit board, with an address of 50 hex and a board index of 0, using the Aardvark adaptor.

Ensure that the Aardvark adaptor is installed so that you can use the i2c interface, and then look at the adaptor properties.

```
instrhwinfo('i2c')
instrhwinfo('i2c', 'Aardvark')
```

Construct an `i2c` object called `I` using `Vendor aardvark`, with `BoardIndex` of `0`, and `RemoteAddress` of `50h`. Note that to specify the remote address of 50 hex, you need to use the `hex2dec` function as shown.

```
I = i2c('aardvark',0,hex2dec('50'));
```

Connect to the chip.

```
fopen(I);
```

Write `'Hello World!'` to the EEPROM chip. Data is written page-by-page in I2C. Each page contains eight bytes. The page address needs to be mentioned before every byte of data written.

The first byte of the string `'Hello World!'` is `'Hello Wo'`. Its page address is 0.

```
fwrite(I,[0 'Hello Wo']);
```

The second byte of the string `'Hello World!'` is `'rld!'`. Its page address is 8.

```
fwrite(I,[8 'rld!']);
```

A zero needs to be written to the `i2c` object, to start reading from the first byte of first page.

```
fwrite(I,0);
```

Read data back from the chip using the `fread` function. The chip returns the characters it was sent.

```
char(fread(I,16))'
```

Disconnect the I2C device.

```
 fclose(I);
```

Clean up by clearing the object.

```
 clear('I');
```

NI-845x Example

This example shows how to communicate with a sensor chip on a circuit board, with an address of 53 hex and a board index of 0, using the NI-845x adaptor. In this case, the NI-845x adaptor board is plugged into the computer (via the USB port), and a circuit board containing the sensor chip is connected to the host adaptor board via wires.

Ensure that the NI-845x adaptor is installed so that you can use the `i2c` interface, and then look at the adaptor properties.

```
instrhwinfo('i2c')
instrhwinfo('i2c', 'NI845x')
```

Construct an `i2c` object called `i2cobj` using `Vendor NI845x`, with `BoardIndex` of `0`, and `RemoteAddress` of `53h`.

```
i2cobj = i2c('NI845x', 0, '53h');
```

Connect to the chip.

```
fopen(i2cobj)
```

Write to the sensor chip. You need to read the documentation or data sheet of the chip in order to know what the remote address is and other information about the chip. In this case, the chip's registry can be opened by sending it a `0`.

```
fwrite(i2cobj, 0)
```

Read data back from the chip using the `fread` function. By sending it one byte, you can read back the device ID registry. In the case of this chip, the read-only device ID registry is 229. Therefore, that is what is returned when you send the byte.

```
fread(i2cobj, 1)

ans =

    229
```

Disconnect the I2C device.

```
fclose(i2cobj);
```

Clean up by deleting and clearing the object.

```
delete(i2cobj);
clear('i2cobj');
```

See Also

"I2C Interface Overview" on page 9-2

icdevice

Create device object

Syntax

```
obj = icdevice('driver', hwobj)
obj = icdevice('driver', 'RsrcName')
obj = icdevice('driver')
obj = icdevice('driver', hwobj, 'P1', V1, 'P2', V2,...)
obj = icdevice('driver', 'RsrcName','P1', V1, 'P2', V2,...)
obj = icdevice('driver','P1', V1, 'P2', V2,...)
```

Arguments

`driver`	A MATLAB instrument driver.
`hwobj`	An interface object.
`RsrcName`	VISA resource name.
`'P1', 'P2',...`	Device-specific property names.
`V1, V2,...`	Property values supported by corresponding *P1*, *P2*,....
`obj`	A device object.

Description

`obj = icdevice('driver', hwobj)` creates the device object `obj`. The instrument-specific information is defined in the MATLAB interface instrument driver, `driver`. Communication to the instrument is done through the interface object, `hwobj`. The interface object can be a serial port, GPIB, VISA, TCPIP, or UDP object. If `driver` does not exist or if `hwobj` is invalid, the device object is not created.

Device objects may also be used with VXI*plug&play* and Interchangeable Virtual Instrument (IVI) drivers. To use these drivers, you must first have a MATLAB instrument driver wrapper for the underlying VXI*plug&play* or IVI driver. If the

MATLAB instrument driver wrapper does not already exist, it may be created using `makemid` or `midedit`. Note that `makemid` or `midedit` only needs to be used once to create the MATLAB instrument driver wrapper.

`obj = icdevice('driver', 'RsrcName')` creates a device object `obj`, using the MATLAB instrument driver, `driver`. The specified `driver` must be a MATLAB VXI*plug&play* instrument driver or MATLAB IVI instrument driver. Communication to the instrument is done through the resource specified by `rsrcname`. For example, all VXI*plug&play*, and many IVI drivers require VISA resource names for `rsrcname`.

`obj = icdevice('driver')` constructs a device object `obj`, using the MATLAB instrument driver, `driver`. The specified `driver` must be a MATLAB IVI instrument driver, and the underlying IVI driver must be referenced using a logical name.

`obj = icdevice('driver', hwobj, 'P1', V1, 'P2', V2,...)`, `obj = icdevice('driver', 'RsrcName','P1', V1, 'P2', V2,...)`, and `obj = icdevice('driver','P1', V1, 'P2', V2,...)`, construct a device object, `obj`, with the specified property values. If an invalid property name or property value is specified, the object will not be created.

Note that the parameter-value pairs can be in any format supported by the `set` function: parameter-value 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:

```
d = icdevice('tektronix_tds210',g,'ObjectVisibility','on');
d = icdevice('tektronix_tds210',g,'objectvisibility','on');
d = icdevice('tektronix_tds210',g,'ObjectVis','on');
```

Note About Deploying Code

When using IVI-C or VXI Plug&Play drivers, executing your code will generate additional file(s) in the folder specified by executing the following code at the MATLAB prompt:

```
fullfile(tempdir,'ICTDeploymentFiles',sprintf('R%s',version('-release')))
```

On all supported platforms, a file with the name `MATLABPrototypeFor<driverName>.m` is generated, where `<driverName>` the name of the IVI-C or VXI Plug&Play driver. With 64-bit MATLAB on Windows, a second file

by the name `<driverName>_thunk_pcwin64.dll` is generated. When creating your deployed application or shared library, manually include these generated files. If using the `icdevice` function, remember to also manually include the MDD-file in the deployed application or shared library. For more information on including additional files refer to the MATLAB Compiler documentation.

Examples

The first example creates a device object for a Tektronix TDS 210 oscilloscope that is connected to a Keithley GPIB board, using a MATLAB interface object and MATLAB interface instrument driver.

```
g = gpib('keithley',0,2);
d = icdevice('tektronix_tds210',g);
```

Connect to the instrument.

```
connect(d);
```

List the oscilloscope settings that can be configured.

```
props = set(d);
```

Get the current configuration of the oscilloscope.

```
values = get(d);
```

Disconnect from the instrument and clean up.

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

The second example creates a device object for a Tektronix TDS 210 oscilloscope using a MATLAB VXI*plug&play* instrument driver.

This example assumes that the `'tktds5k'` VXI*plug&play* driver is installed on your system.

This first step is necessary only if a MATLAB VXI*plug&play* instrument driver for the tktds5k does not exist on your system.

```
makemid('tktds5k', 'Tktds5kMATLABDriver');
```

Construct a device object that uses the VXI*plug&play* driver. The instrument is assumed to be located at GPIB primary address 2.

```
d = icdevice('Tktds5kMATLABDriver', 'GPIB0::2::INSTR');
```

Connect to the instrument.

```
connect(d);
```

List the oscilloscope settings that can be configured.

```
props = set(d);
```

Get the current configuration of the oscilloscope.

```
values = get(d);
```

Disconnect from the instrument and clean up.

```
disconnect(d);
delete(d);
```

More About

Tips

At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with device objects.

```
instrhelp icdevice
```

When you create a device object, these property values are automatically configured:

- `Interface` specifies the interface used to communicate with the instrument. For device objects created using interface objects, it is that interface object. For VXI*plug&play* and IVI-C, this is the session handle to the driver session. For 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 VXI*plug&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`.

Note: ICDEVICE is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

See Also
`connect` | `disconnect` | `instrhelp` | `Status`

inspect

Open Property Inspector

Syntax

```
inspect(obj)
```

Arguments

`obj`	An instrument object or an array of instrument objects.

Description
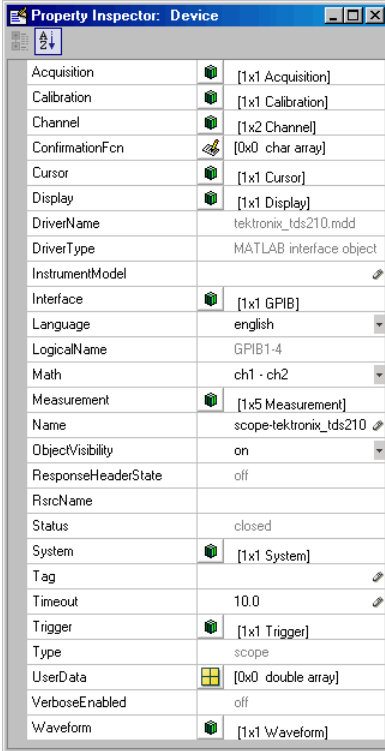
`inspect(obj)` opens the Property Inspector and allows you to inspect and set properties for instrument object `obj`.

More About

Tips

You can also open the Property Inspector via the Workspace browser by right-clicking an instrument object and selecting **Call Property Inspector** from the context menu, or by double-clicking the object.

Below is a Property Inspector for a device object that communicates with a Tektronix TDS 210 oscilloscope.

instrcallback

Display event information when event occurs

Syntax

```
instrcallback(obj, event)
```

Arguments

`obj`	An instrument object.
`event`	The event that caused the callback to execute.

Description

`instrcallback(obj, event)` displays a message that contains the event type, the time the event occurred, and the name of the instrument object that caused the event to occur.

For error events, the error message is also displayed. For pin status events, the pin that changed value and its value are also displayed. For trigger events, the trigger line is also displayed. For datagram received events, the number of bytes received and the datagram address and port are also displayed.

Note: Using this callback for numbers greater than 127 with a terminator is not supported.

Examples

The following example creates the serial port objects `s` on a Windows machine, and configures `s` to execute `instrcallback` when an output-empty event occurs. The event occurs after the `*IDN?` command is written to the instrument.

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

```
set(s,'OutputEmptyFcn',@instrcallback)
fopen(s)
fprintf(s,'*IDN?','async')
```

The resulting display from `instrcallback` is shown below.

```
OutputEmpty event occurred at 08:37:49 for the object: Serial-COM1
```

Read the identification information from the input buffer and end the serial port session.

```
idn = fscanf(s);
fclose(s)
delete(s)
clear s
```

More About

Tips

You should use `instrcallback` as a template from which you create callback functions that suit your specific application needs.

Note: Using this callback for numbers greater than 127 with a terminator is not supported.

---

instrfind

Read instrument objects from memory to MATLAB workspace

Syntax

```
out = instrfind
out = instrfind('PropertyName',PropertyValue,...)
out = instrfind(S)
out = instrfind(obj,'PropertyName',PropertyValue,...)
```

Arguments

`'PropertyName'`	A property name for `obj`.
`PropertyValue`	A property value supported by *PropertyName*.
`S`	A structure of property names and property values.
`obj`	An instrument object, or an array of instrument objects.
`out`	An array of instrument objects.

Description

`out = instrfind` returns all valid instrument objects as an array to `out`.

`out = instrfind('PropertyName',PropertyValue,...)` returns an array of instrument objects whose property names and property values match those specified.

`out = instrfind(S)` returns an array of instrument objects whose property names and property values match those defined in the structure `S`. The field names of `S` are the property names, while the field values are the associated property values.

`out = instrfind(obj,'PropertyName',PropertyValue,...)` restricts the search for matching property name/property value pairs to the instrument objects listed in `obj`.

Examples

Suppose you create the following two GPIB objects.

```
g1 = gpib('ni',0,1);
g2 = gpib('ni',0,2);
g2.EOSCharCode = 'CR';
fopen([g1 g2])
```

You can use `instrfind` to return instrument objects based on property values.

```
out1 = instrfind('Type','gpib');
out2 = instrfind({'Type','EOSCharCode'},{'gpib','CR'});
```

You can also use `instrfind` to return cleared instrument objects to the MATLAB workspace.

```
clear g1 g2
newobjs = instrfind

Instrument Object Array
  Index:  Type:         Status:    Name:
  1       gpib          open       GPIB0-1
  2       gpib          open       GPIB0-2
```

Assign the instrument objects their original names.

```
g1 = newobjs(1);
g2 = newobjs(2);
```

Close both `g1` and `g2`.

```
fclose(newobjs)
```

More About

Tips

`instrfind` will not return an instrument object if its `ObjectVisibility` property is configured to `off`.

You must specify property values using the same format property requires. For example, if the `Name` property value is specified as `MyObject`, `instrfind` will not find an object

with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of 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`.

See Also
clear | instrfindall | propinfo | ObjectVisibility | get

instrfindall

Find visible and hidden instrument objects

Syntax

```
out = instrfindall
out = instrfindall('P1',V1,...)
out = instrfindall(s)
out = instrfindall(objs,'P1',V1,...)
```

Arguments

`'P1'`	Name of an instrument object property or device group object property
`V1`	Value allowed for corresponding *P1*.
`s`	A structure of property names and property values.
`objs`	An array of instrument objects or device group objects.
`out`	An array of returned instrument objects or device group objects.

Description

`out = instrfindall` finds all instrument objects and device group objects, regardless of the value of the objects' `ObjectVisibility` property. The object or objects are returned to `out`.

`out = instrfindall('P1',V1,...)` returns an array, `out`, of instrument objects and device group objects whose property names and corresponding property values match those specified as arguments.

`out = instrfindall(s)` returns an array, `out`, of instrument objects whose property names and corresponding property values match those specified in the structure `s`, where the field names correspond to property names and the field values correspond to the current value of the respective property.

`out = instrfindall(objs,'`*`P1`*`',V1,...)` restricts the search for objects with matching property name/value pairs to the instrument objects and device group objects listed in `objs`.

Note that you can use string property name/property value pairs, structures, and cell array property name/property value pairs in the same call to `instrfindall`.

Examples

Suppose you create the following instrument objects on a Windows machine.

```
s1 = serial('COM1');
s2 = serial('COM2');
g1 = gpib('keithley',0,2);
g1.ObjectVisibility = 'off'
```

Because object `g1` has its `ObjectVisibility` set to off, it is not visible to commands like `instrfind`:

```
instrfind
```

```
 Instrument Object Array
 Index:   Type:           Status:      Name:
 1        serial          closed       Serial-COM1
 2        serial          closed       Serial-COM2
```

However, `instrfindall` finds all objects regardless of the value of `ObjectVisibility`:

```
instrfindall
```

```
 Instrument Object Array
 Index:   Type:           Status:      Name:
 1        serial          closed       Serial-COM1
 2        serial          closed       Serial-COM2
 3        gpib            closed       GPIB0-2
```

The following statements use `instrfindall` to return objects with specific property settings, which are passed as cell arrays:

```
props = {'PrimaryAddress','SecondaryAddress};
vals = {2,0};
```

```
obj = instrfindall(props,vals);
```

You can use `instrfindall` as an argument when you want to apply the command to all objects, visible and invisible. For example, the following statement makes all objects visible:

```
set(instrfindall,'ObjectVisibility','on')
```

More About

Tips

`instrfindall` differs from `instrfind` in that it finds objects whose `ObjectVisibility` property is set to `off`.

Property values are case sensitive. You must specify property values using the same format as that the property requires. For example, if `Name` property value is specified as `MyObject`, `instrfindall` will not find an object with a `Name` property value of `myobject`. However, this is not the case for properties that have a finite set of 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.

See Also
`instrfind` | `propinfo` | `ObjectVisibility`

instrhelp

Help for instrument object type, function, or property

Syntax

```
instrhelp
instrhelp('name')
out = instrhelp('name')
instrhelp(obj)
instrhelp(obj,'name')
out = instrhelp(obj,'name')
```

Arguments

`'name'`	A function name, property name, or instrument object type.
`obj`	An instrument object.
`out`	The help text.

Description

`instrhelp` returns a complete listing of toolbox functions, with a brief description of each.

`instrhelp('name')` returns help for the function, property, or instrument object type specified by *name*.

You can return specific instrument object information by specifying *name* in the form `object/function` or `object.property`. For example, to return the help for a serial port object's `fprintf` function, *name* would be `serial/fprintf`. To return the help for a serial port object's `Parity` property, *name* would be `serial.parity`.

`out = instrhelp('name')` returns the help text to `out`.

`instrhelp(obj)` returns a complete listing of functions and properties for `obj`, with a brief description of each. Help for the constructor is also returned.

`instrhelp(obj,'`*name*`')` returns help for the function or property specified by *name* associated with `obj`.

`out = instrhelp(obj,'`*name*`')` returns the help text to `out`.

Examples

The following commands illustrate some of the ways you can get function and property help without creating an instrument object.

```
instrhelp gpib
out = instrhelp('gpib.m');
instrhelp set
instrhelp('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');
```

More About

Tips

When returning property help, the names in the See Also section that contain all uppercase letters are function names. The names that contain a mixture of upper and lowercase letters are property names. When returning function help, the See Also section contains only function names.

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

See Also
propinfo

instrhwinfo

Information about available hardware

Syntax

```
out = instrhwinfo
out = instrhwinfo('interface')
out = instrhwinfo('drivertype')
out = instrhwinfo('interface','adaptor')
out = instrhwinfo('drivertype','drivername')
out = instrhwinfo('ivi','LogicalName')
out = instrhwinfo('interface','adaptor','type')
out = instrhwinfo(obj)
out = instrhwinfo(obj,'FieldName')
```

Arguments

`'interface'`	A supported instrument interface.
`'drivertype'`	Instrument driver type, may be `matlab`, `ivi`, or `vxipnp`.
`'adaptor'`	A supported GPIB or VISA adaptor.
`'drivername'`	Name of `ivi`, VXI*plug&play*, or MATLAB instrument driver.
`'LogicalName'`	IVI logical name value.
`'type'`	Type of VISA interface.
`obj`	An instrument object or array of instrument objects.
`'FieldName'`	A field name or cell array of field names associated with `obj`.
`out`	A structure or array containing hardware information.

Description

`out = instrhwinfo` returns hardware information to the structure `out`. This information includes the toolbox version, the MATLAB software version, and supported interfaces.

`out = instrhwinfo('`*`interface`*`')` returns information related to the interface specified by *interface*. *interface* can be `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 software path. If *drivertype* is `vxipnp`, this information includes the found VXI*plug&play* drivers. If *drivertype* is `ivi`, this information includes the available logical names and information on the IVI configuration store. You can use either an IVI-C or an IVI-COM driver.

`out = instrhwinfo('`*`interface`*`','`*`adaptor`*`')` returns information related to the adaptor specified by *adaptor*, and for the interface specified by *interface*. *interface* can be `gpib` or `visa`. The returned information includes the adaptor version and available hardware. The GPIB adaptors are `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.

Examples

The following commands illustrate some of the ways you can get hardware-related information without creating an instrument object.

```
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.

```
vs = visa('agilent','ASRL1::INSTR');
out5 = instrhwinfo(vs)
out5 =
           AdaptorDllName: [1x67 char]
        AdaptorDllVersion: 'Version 1.2 (R13)'
              AdaptorName: 'AGILENT'
  VendorDriverDescription: 'Agilent Technologies VISA Driver'
      VendorDriverVersion: '1.1000'

vsdll = instrhwinfo(vs,'AdaptorDllName')
vsdll = D:\V6\toolbox\instrument\instrumentadaptors\win32\
mwagilentvisa.dll
```

More About

Tips

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

instrid

Define and retrieve commands that identify instruments

Syntax

```
instrid
instrid('cmd')
out = instrid(...)
```

Arguments

cmd	The instrument identification command.
out	The list of commands used to locate and identify instruments.

Description

instrid returns the currently defined instrument identification commands.

instrid('cmd') defines the instruments identification commands to be the string cmd. Note that you can also specify a cell array of commands.

out = instrid(...) returns the instrument identification commands to out.

Examples

Set the identification command to *ID?.

```
instrid('*ID?')
```

Specify three new identification commands using a cell array.

```
instrid({'*IDN?','*ID?','IDEN?'})
```

Assign a list of current identification commands to an output variable.

```
id_commands = instrid;
```

More About

Tips

The Instrument Control Toolbox `instrhwinfo` and `tmtool` functions use the instrument identification commands as defined by `instrid` when locating and identifying instruments.

By default, Instrument Control Toolbox software uses the command `*IDN?`, which identifies most instruments. However, some instruments respond to different identification commands such as `*ID?` or `*IDEN?`.

If `instrhwinfo` or `tmtool` does not identify a known instrument, use `instrid` to specify the identification commands the instrument will respond to. If `instrid` returns no commands, an instrument cannot be found.

See Also
`instrhwinfo` | `tmtool`

instrnotify

Define notification for instrument events

Syntax

```
instrnotify('Type', callback)
instrnotify({'P1', 'P2', ...}, 'Type', callback)
instrnotify(obj, 'Type', callback)
instrnotify(obj, {'P1', 'P2', ...}, 'Type', callback)
instrnotify('Type', callback, '-remove')
instrnotify(obj, 'Type', callback, '-remove')
```

Arguments

`'Type'`	The type of event: `ObjectCreated`, `ObjectDeleted`, or `PropertyChangedPostSet`
`callback`	Function handle, 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

```
function(obj, event, [arg1, arg2,...])
```

where `obj` is the instrument object or device group object generating the event. `event` is a structure containing information on the event generated. If *Type* is `ObjectCreated` or `ObjectDeleted`, `event` contains the type of event. If *Type* is `PropertyChangedPostSet`, `event` contains the type of event, the property being configured, and the new property value.

`instrnotify({'P1', 'P2', ...}, 'Type', callback)` evaluates the MATLAB expression, `callback`, in the MATLAB workspace when any of the specified properties, *P1*, *P2*, ... are configured. *Type* can be only `PropertyChangedPostSet`.

`instrnotify(obj, 'Type', callback)` evaluates the MATLAB expression, `callback`, in the MATLAB workspace when an event of type *Type* for object `obj`, is generated. `obj` can be an array of instrument objects or device group objects.

`instrnotify(obj, {'P1', 'P2', ...}, 'Type', callback)` evaluates the MATLAB expression, `callback`, in the MATLAB workspace when any of the specified properties, *P1*, *P2*, are configured on object `obj`.

`instrnotify('Type', callback, '-remove')` removes the specified `callback` of type *Type*.

`instrnotify(obj, 'Type', callback, '-remove')` removes the specified `callback` of type *Type* for object `obj`.

Examples

```
instrnotify('PropertyChangedPostSet', @instrcallback);
g = gpib('keithley', 0, 5);
set(g, 'Name', 'mygpib');
fopen(g);
fclose(g);
```

```
instrnotify('PropertyChangedPostSet',@instrcallback,'-remove');
```

More About

Tips

PropertyChangedPostSet events are generated only when the property is configured to a different value than what the property is currently configured to. For example, if a GPIB object's Tag property is configured to 'myobject', a PropertyChangedPostSet event will not be generated if the object's Tag property is currently set to 'myobject'. A PropertyChangedPostSet event will be generated if the object's Tag property is set to 'myGPIBObject'.

If obj is specified and the callback *Type* is ObjectCreated, the callback will not be generated because obj has already been created.

If *Type* is ObjectDeleted, the invalid object obj is not passed as the first input argument to the callback function. Instead, an empty matrix is passed as the first input argument.

instrreset

Disconnect and delete all instrument objects

Syntax

```
instrreset
```

Description

`instrreset` disconnects and deletes all instrument objects.

More About

Tips

If data is being written or read asynchronously, the asynchronous operation is stopped.

`instrreset` is equivalent to issuing the `stopasync` (if needed), `fclose`, and `delete` functions for all instrument objects.

When you delete an instrument object, it becomes *invalid*. Because you cannot connect an invalid object to the instrument, you should remove it from the workspace with the `clear` command.

See Also
`clear` | `delete` | `fclose` | `isvalid` | `stopasync`

invoke

Execute driver-specific function on device object

Syntax

```
out = invoke(obj,'name')
out = invoke(obj,'name',arg1,arg2,...)
```

Arguments

obj	A device object.
name	The function to execute.
arg1,arg2,...	Arguments passed to name.
out	The function output.

Description

out = invoke(obj,'name') executes the function specified by name on the device object specified by obj. The function's output is returned to out.

out = invoke(obj,'name',arg1,arg2,...) passes the arguments arg1,arg2,... to the function specified by name.

Examples

Create a device object for a Tektronix TDS 210 oscilloscope that is connected to a National Instruments GPIB board.

```
g = gpib('ni',0,2);
d = icdevice('tektronix_tds210',g);
```

Perform a self-calibration for the oscilloscope by invoking the calibrate function.

```
out = invoke(d,'calibrate')
out =
    '0'
```

0 indicates that the self-calibration completed without any errors.

More About

Tips

To list the driver-specific functions supported by obj, type

```
methods(obj)
```

To display help for a specific function, type

```
instrhelp(obj,'name')
```

See Also
instrhelp | methods | Status

isvalid

Determine whether instrument objects are valid

Syntax

```
out = isvalid(obj)
```

Arguments

obj	An instrument object or array of instrument objects.
out	A logical array.

Description

`out = isvalid(obj)` returns the logical array `out`, which contains a 0 where the elements of `obj` are invalid instrument objects and a 1 where the elements of `obj` are valid instrument objects.

Examples

Suppose you create the following two GPIB objects:

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

`g2` becomes invalid after it is deleted.

```
delete(g2)
```

`isvalid` verifies that `g1` is valid and `g2` is invalid.

```
garray = [g1 g2];
isvalid(garray)
ans =
```

        1   0

More About

Tips

obj becomes invalid after it is removed from memory with the delete function. Because you cannot connect an invalid object to the instrument, you should remove it from the workspace with the clear command.

See Also
clear | delete

iviconfigurationstore

Create IVI configuration store object

Syntax

```
obj = iviconfigurationstore
obj = iviconfigurationstore('file')
```

Arguments

`obj`	IVI configuration store object
`'file'`	Configuration store data file

Description

`obj = iviconfigurationstore` creates an IVI configuration store object and establishes a connection to the IVI Configuration Server. The data in the master configuration store is used.

`obj = iviconfigurationstore('file')` creates an IVI configuration store object and establishes a connection to the IVI Configuration Server. The data in the configuration store, `file`, is used. If `file` cannot be found or is not a valid configuration store, an error occurs.

See Also
add | commit | remove | update

length

Length of instrument object array

Syntax

```
length(obj)
```

Arguments

obj   An instrument object or an array of instrument objects.

Description

length(obj) returns the length of obj. It is equivalent to the command max(size(obj)).

See Also
instrhelp | size

load

Load instrument objects and variables into MATLAB workspace

Syntax

```
load filename
load filename obj1 obj2 ...
out = load('filename','obj1','obj2',...)
```

Arguments

`filename`	The MAT-file name.
`obj1 obj2 ...`	Instrument objects or arrays of instrument objects.
`out`	A structure containing the specified instrument objects.

Description

`load filename` returns all variables from the MAT-file specified by `filename` into the MATLAB workspace.

`load filename obj1 obj2 ...` returns the instrument objects specified by `obj1 obj2...` from the MAT-file `filename` into the MATLAB workspace.

`out = load('filename','obj1','obj2',...)` returns the specified instrument objects from the MAT-file `filename` as a structure to `out` instead of directly loading them into the workspace. The field names in `out` match the names of the loaded instrument objects.

Examples

Suppose you create the GPIB objects `g1` and `g2`, configure a few properties for `g1`, and connect both objects to their associated instruments.

```
g1 = gpib('ni',0,1);
g2 = gpib('ni',0,2);
set(g1,'EOSMode','read','EOSCharCode','CR')
fopen([g1 g2])
```

The read-only `Status` property is automatically configured to `open`.

```
g1.Status
ans =
    open

g2.Status
ans =
    open
```

Save `g1` and `g2` to the file `MyObject.mat`, and then load the objects into the MATLAB workspace.

```
save MyObject g1 g2
load MyObject g1 g2
```

Values for read-only properties are restored to their default values upon loading, while all other property values are honored.

```
get([g1 g2],{'EOSMode','EOSCharCode','Status'})
ans =
    'read'      'CR'      'closed'
    'none'      'LF'      'closed'
```

More About

Tips

Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages or use the `propinfo` function.

See Also

`instrhelp` | `propinfo` | `save`

makemid

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 VXI*plug&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 VXI*plug&play* instrument driver name. The MATLAB instrument driver will be saved in the current working directory as `driver.mdd`

The MATLAB instrument driver can then be modified using `midedit` to customize the driver behavior, and may be used to instantiate a device object using `icdevice`.

makemid('driver', 'filename') creates and saves the MATLAB instrument driver using the name and path specified by filename.

makemid('driver', '*type*') and makemid('driver', 'filename', '*type*') override the default search order and look only for drivers whose type is *type*. Valid types are vxiplug&play, 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.

Note: MAKEMID is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

Examples

To convert the driver hp34401 into the MATLAB instrument driver hp34401.mdd in the current working directory,

```
makemid('hp34401');
```

To convert the driver tktds5k into the MATLAB instrument driver with a specific name and location,

```
makemid('tktds5k', 'C:\MyDrivers\tektronix_5k.mdd');
```

To convert the IVI-C driver tktds5k into the MATLAB instrument driver tktds5k.mdd in the current working directory. This example causes the function to look for the driver only among the IVI-C drivers.

```
makemid('tktds5k', 'ivi-c');
```

To create the MATLAB instrument driver MyIviLogicalName.mdd from the IVI logical name MyIviLogicalName,

```
makemid('MyIviLogicalName');
```

See Also
icdevice | midedit

memmap

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).

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)
```

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)
```

More About

Tips

Before you can map memory, obj must be connected to the instrument with the fopen function. A connected interface object has a Status property value of open. An error is returned if you attempt to map memory while obj is not connected to the instrument.

To unmap the memory, use the memunmap function. If memory is mapped and fclose is called, the memory is unmapped before the object is disconnected from the instrument.

The MappedMemorySize property returns the size of the memory space mapped. You must map the memory space before using the mempoke or mempeek function.

See Also
fopen | fclose | mempeek | mempoke | memunmap | MappedMemorySize | Status

mempeek

Low-level memory read from VXI register

Syntax

```
out = mempeek(obj,offset)
out = mempeek(obj,offset,'precision')
```

Arguments

`obj`	A VISA-VXI or VISA-GPIB-VXI object.
`offset`	The offset in the mapped memory space from which the data is read.
`'precision'`	The number of bits to read from the memory address.
`out`	An array containing the returned value.

Description

`out = mempeek(obj,offset)` reads a `uint8` value from the mapped memory space specified by `offset` for the object `obj`. The value is returned to `out`.

`out = mempeek(obj,offset,'precision')` reads the number of bits specified by *precision*, from the mapped memory space specified by `offset`. *precision* can be `uint8`, `uint16`, or `uint32`, which instructs `mempeek` to read 8-, 16-, or 32-bit values, respectively. *precision* can also be `single`, which instructs `mempeek` to read single precision values.

Examples

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

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

Use `memmap` to map 16 bytes in the A16 address space.

```
memmap(vv,'A16',0,16)
```

Perform a low-level read of the first and second instrument registers.

```
reg1 = mempeek(vv,0,'uint16')
reg1 =
        53247
reg2 = mempeek(vv,2,'uint16')
reg2 =
        20993
```

Unmap the memory and disconnect `vv` from the instrument.

```
memunmap(vv)
fclose(vv)
```

Refer to "Using High-Level Memory Functions" on page 5-16 for a description of the first four registers of the E1432A digitizer.

More About

Tips

Before you can read from the VXI register, `obj` must be connected to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt a read operation while `obj` is not connected to the instrument.

You must map the memory space using the `memmap` function before using `mempeek`. The `MappedMemorySize` property returns the size of the memory space mapped.

`offset` indicates the offset in the mapped memory space from which the data is read. For example, if the mapped memory space begins at 200H, the offset is 2, and the precision is `uint8`, then the data is read from memory location 202H. If the precision is `uint16`, the data is read from 202H and 203H.

To increase speed, `mempeek` does not return error messages from the instrument.

See Also

`fopen` | `memmap` | `mempoke` | `memunmap` | `MappedMemorySize` | `MemoryIncrement` | `Status`

mempoke

Low-level memory write to VXI register

Syntax

```
mempoke(obj,data,offset)
mempoke(obj,data,offset,'precision')
```

Arguments

obj	A VISA-VXI or VISA-GPIB-VXI object.
data	The data written to the memory address.
offset	The offset in the mapped memory space to which the data is written.
'*precision*'	The number of bits to write to the memory address.

Description

mempoke(obj,data,offset) writes the uint8 value specified by data to the mapped memory address specified by offset for the object obj.

mempoke(obj,data,offset,'precision') writes data using the number of bits specified by *precision*. *precision* can be uint8, uint16, or uint32, which instructs mempoke to write data as 8-, 16-, or 32-bit values, respectively. *precision* can also be single, which instructs mempoke to write data as single-precision values.

Examples

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

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

```
fopen(vv)
```

Use `memmap` to map 16 bytes in the A16 address space.

```
memmap(vv,'A16',0,16)
```

Perform a low-level write to the fourth instrument register, which has an offset of 6.

```
mempoke(vv,45056,6,'uint16')
```

Unmap the memory and disconnect `vv` from the instrument.

```
memunmap(vv)
fclose(vv)
```

Refer to "Using High-Level Memory Functions" on page 5-16 for a description of the first four registers of the E1432A digitizer.

More About

Tips

Before you can write to the VXI register, `obj` must be connected to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt a write operation while `obj` is not connected to the instrument.

You must map the memory space using the `memmap` function before using `mempoke`. The `MappedMemorySize` property returns the size of the memory space mapped.

`offset` indicates the offset in the mapped memory space to which the data is written. For example, if the mapped memory space begins at 200H, the offset is 2, and the precision is `uint8`, then the data is written to memory location 202H. If the precision is `uint16`, the data is written to 202H and 203H.

To increase speed, `mempoke` does not return error messages from the instrument.

See Also
fopen | memmap | mempeek | MappedMemorySize | MemoryIncrement | Status

memread

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.

out = memread(obj,offset,'*precision*','*adrspace*',size) reads a block of data with a size specified by size.

Examples

Create the VISA-VXI object vv associated with a VXI chassis with index 0, and 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 "Using High-Level Memory Functions" on page 5-16 for a description of the first four registers of the E1432A digitizer.

More About

Tips

Before you can read data from the VXI register, obj must be connected to the instrument with the fopen function. A connected interface object has a Status property value of

open. An error is returned if you attempt to read memory while obj is not connected to the instrument.

See Also
fopen | mempeek | memwrite | MemoryIncrement | MemorySpace | Status

memunmap

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.

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)
```

Map 16 bytes in the A16 address space.

```
memmap(vv,'A16',0,16)
```

Read the first and second instrument registers.

```
reg1 = mempeek(vv,0,'uint16');
reg2 = mempeek(vv,2,'uint16');
```

Use memunmap to unmap the memory, and disconnect vv from the instrument.

```
memunmap(vv)
```

```
fclose(vv)
```

More About

Tips

When the memory space is unmapped, the MappedMemorySize property is set to 0 and the MappedMemoryBase property is set to 0H.

See Also

memmap | mempeek | mempoke | MappedMemoryBase | MappedMemorySize

memwrite

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.

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 write to the fourth instrument register, which has an offset of 6.

```
memwrite(vv,45056,6,'uint16','A16')
```

Disconnect vv from the instrument.

```
fclose(vv)
```

Refer to "Using High-Level Memory Functions" on page 5-16 for a description of the first four registers of the E1432A digitizer.

More About

Tips

Before you can write to the VXI register, obj must be connected to the instrument with the fopen function. A connected interface object has a Status property value of open. An error is returned if you attempt a write operation while obj is not connected to the instrument.

See Also
fopen | memread | mempoke | MemoryIncrement | MemorySpace | Status

methods

Class method names and descriptions

Syntax

```
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.

More About

Tips

methods differs from what in that the methods from all method directories are reported together, and methods removes all duplicate method names from the result list. methods will also return the methods for a Java class.

See Also
methodsview | what | which | help

midedit

Open graphical tool for creating and editing MATLAB instrument driver

Syntax

```
midedit
midedit('driver')
```

Arguments

`'driver'`   The name of a MATLAB instrument driver.

Description

`midedit` opens the MATLAB Instrument Driver Editor, which is a graphical tool for creating and editing instrument drivers.

`midedit('driver')` opens the MATLAB Instrument Driver Editor for the specified instrument driver. The default extension for driver is `.mdd`. Note that `driver` can include a relative partial pathname.

The editor consists of two main parts: the navigation pane and the detail pane. The navigation pane lists the driver-specific properties and functions in a tree view, while the detail pane allows you to configure and document the properties and functions.

`midedit` may also be used to import VXI*plug&play* or IVI drivers. With `midedit` open, select **Import** from the **File** menu. The import process creates a new MATLAB Instrument Driver based on the VXI*plug&play* or IVI driver. This allows you to customize the behavior of device objects that use the VXI*plug&play* or IVI driver.

For details and examples on the MATLAB Instrument Driver Editor, see "MATLAB Instrument Driver Editor Overview" on page 18-2.

Note: MIDEDIT is unable to open MDDs with non-ascii characters either in their name or path on Mac platforms.

See Also
icdevice | makemid | midtest | tmtool

midtest

Open graphical tool for testing MATLAB instrument driver

Syntax

```
midtest
midtest('file')
```

Arguments

'file'   File containing the test to be used by the MATLAB Instrument Driver
            Testing Tool

Description

midtest opens the MATLAB Instrument Driver Testing Tool. The MATLAB Instrument
Driver Testing Tool provides a graphical environment for creating a test to verify the
functionality of a MATLAB instrument driver.

The MATLAB Instrument Driver Testing Tool provides a way to

- Verify property behavior
- Verify function behavior
- Save the test as MATLAB code
- Export the test results to MATLAB workspace, figure window, MAT-file, or the
  MATLAB Variables editor
- Save test results as an HTML page

midtest('file') opens the MATLAB Instrument Driver Testing Tool with the test
loaded from file.

For a full description of the tool with examples, see "Instrument Driver Testing Tool
Overview" on page 19-2.

Examples

```
midtest('test.xml')
```

opens the MATLAB Instrument Driver Testing Tool with the test `test.xml` loaded.

See Also
`icdevice` | `makemid` | `midedit` | `tmtool`

obj2mfile

Convert instrument object to MATLAB code

Syntax

```
obj2mfile(obj,'filename')
obj2mfile(obj,'filename','syntax')
obj2mfile(obj,'filename','mode')
obj2mfile(obj,'filename','syntax','mode')
obj2mfile(obj,'filename','reuse')
obj2mfile(obj,'filename','syntax','mode','reuse')
```

Arguments

`obj`	An instrument object or an array of instrument objects.
`'filename'`	The name of the file that the MATLAB code is written to. You can specify the full pathname. If an extension is not specified, the .m extension is used.
`'syntax'`	Syntax of the converted MATLAB code. By default, the `set` syntax is used. If `dot` is specified, then the dot notation is used.
`'mode'`	Specifies whether all properties are converted to code, or only modified properties are converted to code.
`'reuse'`	Specifies whether existing object is reused or new object is created.

Description

`obj2mfile(obj,'filename')` converts `obj` to the equivalent MATLAB code using the `set` syntax and saves the code to `filename`. Only those properties not set to their default value are saved.

`obj2mfile(obj,'filename','syntax')` converts `obj` to the equivalent MATLAB code using the syntax specified by `syntax`. You can specify `syntax` to be `set` or `dot`. `set` uses the set syntax, while `dot` uses the dot notation.

obj2mfile(obj,'filename','*mode*') converts the properties specified by *mode*. You can specify *mode* to be all or modified. If *mode* is all, then all properties are converted to code. If *mode* is modified, then only those properties not set to their default value are converted to code.

obj2mfile(obj,'filename','*syntax*','*mode*') converts the specified properties to code using the specified syntax.

obj2mfile(obj,'filename','*reuse*')

obj2mfile(obj,'filename','*syntax*','*mode*','*reuse*') check for an existing instrument object, obj, before creating obj. If *reuse* is reuse, the object is used if it exists, otherwise the object is created. If *reuse* is create, the object is always created. By default, *reuse* is reuse.

An object will be reused if the existing object has the same constructor arguments as the object about to be created, and if their Type and Tag property values are the same.

Examples

Suppose you create the GPIB object g, and configure several property values.

```
g = gpib('ni',0,1);
set(g,'Tag','MyGPIB object','EOSMode','read','EOSCharCode','CR')
set(g,'UserData',{'test',2,magic(10)})
```

The following command writes MATLAB code to the files MyGPIB.m and MyGPIB.mat.

```
obj2mfile(g,'MyGPIB.m','dot')
```

MyGPIB.m contains code that recreates the commands shown above using the dot notation for all properties that have their default values changed. Because UserData is set to a cell array of values, this property appears in MyGPIB.m as

```
obj1.UserData = userdata1;
```

It is saved in MyGPIB.mat as

```
userdata = {'test', 2, magic(10)};
```

To recreate g in the MATLAB workspace using a new variable, gnew,

```
gnew = MyGPIB;
```

The associated MAT-file, `MyGPIB.mat`, is automatically run and `UserData` is assigned the appropriate values.

```
gnew.UserData
ans =
    'test'    [2]    [10x10 double]
```

More About

Tips

You can recreate a saved instrument object by typing the name of the file at the MATLAB Command Window.

If the `UserData` property is not empty or if any of the callback properties are set to a cell array of values or a function handle, then the data stored in those properties is written to a MAT-file when the instrument object is converted and saved. The MAT-file has the same name as the file containing the instrument object code (see the example below).

Read-only properties are restored with their default values. For example, suppose an instrument object is saved with a `Status` property value of `open`. When the object is recreated, `Status` is set to its default value of `closed`.

See Also
`propinfo`

oscilloscope

Create Quick-Control Oscilloscope object

Syntax

```
myScope = oscilloscope()
connect(myScope);
set(myScope, 'P1',V1,'P2',V2,...)
waveformArray = getWaveform(myScope);
```

Description

The Quick-Control Oscilloscope can be used for any oscilloscope that uses VISA and an underlying IVI-C or IVI-COM driver. However, you do not have to directly deal with the underlying driver. You can also use it for Tektronix oscilloscopes. This is an easy to use oscilloscope object.

`myScope = oscilloscope()` creates an instance of the scope named `myScope`.

`connect(myScope);` connects to the scope.

`set(myScope, 'P1',V1,'P2',V2,...)` assigns the specified property values.

`waveformArray = getWaveform(myScope);` acquires a waveform from the scope.

For information on the prerequisites for using `oscilloscope`, see "Quick-Control Oscilloscope Prerequisites" on page 13-26.

The Quick-Control Oscilloscope `oscilloscope` function can use the following special functions, in addition to standard functions such as `connect` and `disconnect`.

Function	Description
autoSetup	Automatically configures the instrument based on the input signal. `autoSetup(myScope);`
disableChannel	Disables oscilloscope's channel(s).

Function	Description

	``
	disableChannel('Channel1');
	disableChannel({'Channel1', 'Channel2'});
	``

enableChannel	Enables oscilloscope's channel(s) from which waveform(s) will be retrieved.
	
	``
	enableChannel('Channel1');
	enableChannel({'Channel1', 'Channel2'});
	``

getDrivers	Retrieves a list of available oscilloscope instrument drivers. Returns a list of available drivers with their supported instrument models.
	
	``
	drivers = getDrivers(myScope);
	``

getResources	Retrieves a list of available resources of instruments. It returns a list of available VISA resource strings when using an IVI-C or IVI-COM scope. It returns the interface resource information when using a Tektronix scope.
	
	``
	res = getResources(myScope);
	``

getVerticalCoupling	Returns the value of how the oscilloscope couples the input signal for the selected channel name as a MATLAB string. Possible values returned are 'AC', 'DC', and 'GND'. ``` VC = getVerticalCoupling(myScope, 'Channel1'); ```
getVerticalOffset	Returns location of the center of the range for the selected channel name as a MATLAB string. The units are volts. ``` VO = getVerticalOffset(myScope, 'Channel1'); ```
getVerticalRange	Returns absolute value of the input range the oscilloscope can acquire for selected channel name as a MATLAB string. The units are volts. ``` VR = getVerticalRange(myScope, 'Channel1'); ```

getWaveform	Returns the waveform(s) displayed on the scope screen. Retrieves the waveform(s) from enabled channel(s).
	
	``
	w = getWaveform(myScope);
	``

Function	Description
setVerticalCoupling	Specifies how the oscilloscope couples the input signal for the selected channel name as a MATLAB string. Valid values are 'AC', 'DC', and 'GND'. `setVerticalCoupling(myScope, 'Channel1', 'AC');`
setVerticalOffset	Specifies location of the center of the range for the selected channel name as a MATLAB string. For example, to acquire a sine wave that spans between 0.0 and 10.0 volts, set this attribute to 5.0 volts. `setVerticalOffset(myScope, 'Channel1', 5);`
setVerticalRange	Specifies the absolute value of the input range the oscilloscope can acquire for the selected channel name as a MATLAB string. The units are volts. `setVerticalRange(myScope, 'Channel1', 10);`

Arguments

The Quick-Control Oscilloscope `oscilloscope` function can use the following properties.

Property	Description
`ChannelNames`	Read-only property that provides available channel names in a cell array.
`ChannelsEnabled`	Read-only property that provides currently enabled channel names in a cell array.
`Status`	Read-only property that indicates the communication status. Valid values are `open` or `closed`.
`Timeout`	Use to get or set a timeout value. Value cannot be negative number. Default is `10` seconds.
`AcquisitionTime`	Use to get or set acquisition time value. Used to control the time in seconds that corresponds to the record length. Value must be a positive, finite number.
`AcquisitionStartDelay`	Use to set or get the length of time in seconds from the trigger event to first point in waveform record. If positive, the first point in the waveform occurs after the trigger. If negative, the first point in the waveform occurs before the trigger.
`TriggerMode`	Use to set the triggering behavior. Values are: `'normal'` – the oscilloscope waits until the trigger the user specifies occurs. `'auto'` – the oscilloscope automatically triggers if the configured trigger does not occur within the oscilloscope's timeout period.
`TriggerSlope`	Use to set or get trigger slope value. Valid values are `falling` or `rising`.
`TriggerLevel`	Specifies the voltage threshold in volts for the trigger control.

Property	Description
TriggerSource	Specifies the source the oscilloscope monitors for a trigger. It can be channel name or other values.
Resource	Set up before connecting to instrument. Set with value of your instrument's resource string, for example: ```set(myScope, 'Resource', 'TCPIP0::a-m6104a-004598::inst0::INSTR');```
DriverDetectionMode	Optionally used to set up criteria for connection. Valid values are auto or manual. Default is auto. auto means you do not have to set a driver name before connecting to an instrument. If set to manual, a driver name must be provided before connecting.
Driver	Use only if set DriverDetectionMode to manual. Then use to give driver name. Only use if driver name cannot be figured out programmatically.

Examples

Create an instance of the scope called myScope.

```
myScope = oscilloscope()
```

Discover available resources. A resource string is an identifier to the instrument. You need to set it before connecting to the instrument.

```
availableResources = getResources(myScope)
```

If multiple resources are available, use your VISA utility to verify the correct resource and set it.

```
set(myScope, 'Resource', 'TCPIP0::a-m6104a-004598::inst0::INSTR');
```

Connect to the scope.

```
connect(myScope);
```

Automatically configure the scope based on the input signal.

```
autoSetup(myScope);
```

Configure the oscilloscope.

```
% Set the acquisition time to 0.01 second.
set(myScope, 'AcquisitionTime', 0.01);

% Set the acquisition to collect 2000 data points.
set(myScope, 'WaveformLength', 2000);

% Set the trigger mode to normal.
set(myScope, 'TriggerMode', 'normal');

% Set the trigger level to 0.1 volt.
set(myScope, 'TriggerLevel', 0.1);

% Enable channel 1.
enableChannel(myScope, 'Channel1');

% Set the vertical coupling to AC.
setVerticalCoupling (myScope, 'Channel1', 'AC');

% Set the vertical range to 5.0.
setVerticalRange (myScope, 'Channel1', 5.0);
```

Communicate with the instrument. For example, read a waveform.

```
% Acquire the waveform.
waveformArray = getWaveform(myScope);

% Plot the waveform and assign labels for the plot.
plot(waveformArray);
xlabel('Samples');
ylabel('Voltage');
```

More About

- "Using Quick-Control Oscilloscope" on page 13-26

propinfo

Instrument object property information

Syntax

```
out = propinfo(obj)
out = propinfo(obj,'PropertyName')
```

Arguments

`obj`	An instrument object.
`'PropertyName'`	A property name or cell array of property names.
`out`	A structure containing property information.

Description

`out = propinfo(obj)` returns the structure `out` with field names given by the property names for `obj`. Each property name in `out` contains the fields shown below.

Field Name	Description
Type	The property data type. Possible values are `any`, `ASCII value`, `callback`, `instrument range value`, `double`, `string`, and `struct`.
Constraint	The type of constraint on the property value. Possible values are `ASCII value`, `bounded`, `callback`, `instrument range value`, `enum`, and `none`.
ConstraintValue	Property value constraint. The constraint can be a range of valid values or a list of valid string values.
DefaultValue	The property default value.
ReadOnly	The condition under which a property is read-only. Possible values are `always`, `never`, `whileOpen`, and `whileRecording`.

Field Name	Description
Interface Specific	If the property is interface-specific, a 1 is returned. If a 0 is returned, the property is supported for all interfaces.

out = propinfo(obj,'*PropertyName*') returns the structure out for the property specified by *PropertyName*. The field names of out are given in the table shown above. If *PropertyName* is a cell array of property names, a cell array of structures is returned for each property.

Examples

To return all property information for the GPIB object g,

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

To display all the property information for the InputBufferSize property,

```
out.InputBufferSize
ans =
                Type: 'double'
          Constraint: 'none'
     ConstraintValue: ''
        DefaultValue: 512
            ReadOnly: 'whileOpen'
   InterfaceSpecific: 0
```

To display the default value for the EOSMode property,

```
out.EOSMode.DefaultValue
ans =
none
```

More About

Tips

You can get help for instrument object properties with the instrhelp function.

You can display all instrument object property names and their current values using the `get` function. You can display all configurable properties and their possible values using the `set` function.

When specifying property names, you can do so without regard to case, and you can make use of property name completion. For example, if `g` is a GPIB object, then the following commands are all valid.

```
out = propinfo(g,'EOSMode');
out = propinfo(g,'eosmode');
out = propinfo(g,'EOSM');
```

See Also
instrhelp | set | get

query

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.

Examples

This example creates the GPIB object `g`, connects `g` to a Tektronix TDS 210 oscilloscope, writes and reads text data using `query`, and then disconnects `g` from the instrument.

```
g = gpib('ni',0,1);
fopen(g)
idn = query(g,'*IDN?')
idn =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
fclose(g)
```

More About

Tips

Before you can write or read data, `obj` must be connected to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a query operation while `obj` is not connected to the instrument.

`query` operates only in synchronous mode, and blocks the command line until the write and read operations complete execution.

Using `query` is equivalent to using the `fprintf` and `fgets` functions. The rules for completing a write operation are described in the `fprintf` reference pages. The rules for completing a read operation are described in the `fgets` reference pages.

See Also

`fopen` | `Status` | `fprintf` | `fgets` | `sprintf`

read

Read binary data from SPI instrument

Syntax

```
A = read(OBJ, SIZE)
```

Description

`A = read(OBJ, SIZE)` reads the specified number of values, SIZE, from the SPI device connected to interface object, OBJ, and returns to A. OBJ must be a 1-by-1 SPI interface object. By default the `'uint8'` precision is used.

The interface object must be connected to the device with the `connect` function before any data can be read from the device, otherwise an error is returned. A connected interface object has a `ConnectionStatus` property value of `connected`.

Available options for SIZE include: N — read at most N values into a column vector. SIZE cannot be set to `INF`.

The SPI protocol operates in full duplex mode, input and output data transfers happen simultaneously. SPI communication requires N bytes of dummy data to be written into the device for reading N bytes of data from the device. The dummy data written is zeros.

For more information on using the SPI interface and this function, see "Configuring SPI Communication" on page 10-4 and "Transmitting Data Over the SPI Interface" on page 10-9.

Examples

This example shows how to create a SPI object s, and read data.

Construct a `spi` object called s using `Vendor 'aardvark'`, with `BoardIndex` of `0`, and `Port` of `0`.

```
s = spi('aardvark', 0, 0);
```

Connect to the chip.

```
connect(s);
```

Read data from the chip.

```
data = read(s, 2);
```

Disconnect the SPI device and clean up by clearing the object.

```
disconnect(s);
clear('s');
```

readasync

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.

Examples

This example creates the serial port object `s`, connects `s` to a Tektronix TDS 210 oscilloscope, configures `s` on a Windows machine to read data asynchronously only if `readasync` is issued, and configures the instrument to return the peak-to-peak value of the signal on channel 1.

```
s = serial('COM1');
fopen(s)
s.ReadAsyncMode = 'manual';
fprintf(s,'Measurement:Meas1:Source CH1')
fprintf(s,'Measurement:Meas1:Type Pk2Pk')
fprintf(s,'Measurement:Meas1:Value?')
```

Initially, there is no data in the input buffer.

```
s.BytesAvailable
ans =
     0
```

Begin reading data asynchronously from the instrument using `readasync`. When the read operation is complete, return the data to the MATLAB workspace using `fscanf`.

```
readasync(s)
s.BytesAvailable
ans =
     15
out = fscanf(s)
out =
2.0399999619E0
fclose(s)
```

More About

Tips

Before you can read data, you must connect `obj` to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

For serial port, TCPIP, UDP, and VISA-serial objects, you should use `readasync` only when you configure the `ReadAsyncMode` property to `manual`. `readasync` is ignored if used when `ReadAsyncMode` is `continuous`.

The `TransferStatus` property indicates if an asynchronous read or write operation is in progress. For all interface objects, you cannot use `readasync` while a read operation is in progress. For serial port and VISA-serial objects, you can write data while an asynchronous read is in progress because serial ports have separate read and write pins. You can stop asynchronous read and write operations with the `stopasync` function.

You can monitor the amount of data stored in the input buffer with the `BytesAvailable` property. Additionally, you can use the `BytesAvailableFcn` property to execute a callback function when the terminator or the specified amount of data is read.

Asynchronous operation is not supported for NI VISA objects on the UNIX platform. So if you use the `readasync` function with a NI VISA object, you will get an error.

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.

See Also

`fopen` | `stopasync` | `BytesAvailable` | `BytesAvailableFcn` | `ReadAsyncMode` | `Status` | `TransferStatus`

record

Record data and event information to file

Syntax

```
record(obj)
record(obj,'switch')
```

Arguments

obj	An instrument object.
'*switch*'	Switch recording capabilities on or off.

Description

record(obj) toggles the recording state for obj.

record(obj,'*switch*') initiates or terminates recording for obj. *switch* can be on or off. If *switch* is on, recording is initiated. If *switch* is off, recording is terminated.

Examples

This example creates the GPIB object g, connects g to the instrument, and configures g to record detailed information to the disk file MyGPIBFile.txt.

```
g = gpib('ni',0,1);
fopen(g)
g.RecordDetail = 'verbose';
g.RecordName = 'MyGPIBFile.txt';
```

Initiate recording, write the *IDN? command to the instrument, and read back the identification information.

```
record(g,'on')
```

```
fprintf(g,'*IDN?')
out = fscanf(g);
```

Terminate recording and disconnect g from the instrument.

```
record(g,'off')
fclose(g)
```

More About

Tips

Before you can record information to disk, obj must be connected to the instrument with the fopen function. A connected instrument object has a Status property value of open. An error is returned if you attempt to record information while obj is not connected to the instrument. Each instrument object must record information to a separate file. Recording is automatically terminated when obj is disconnected from the instrument with fclose.

The RecordName and RecordMode properties are read-only while obj is recording, and must be configured before using record.

For a detailed description of the record file format and the properties associated with recording data and event information to a file, refer to "Debugging: Recording Information to Disk" on page 16-6.

See Also

fclose | fopen | propinfo | RecordMode | RecordName | RecordStatus | Status

remove

Remove entry from IVI configuration store object

Syntax

```
remove(obj, 'type', 'name')
remove(obj, struct)
```

Arguments

`obj`	IVI configuration store object
`'type'`	Type of entry being removed; *type* can be `DriverSession`, `HardwareAsset`, or `LogicalName`
`'name'`	Name of the `DriverSession`, `HardwareAsset`, or `LogicalName` to be removed
`struct`	Structure defining entries to be removed

Description

`remove(obj, 'type', 'name')` removes an entry of type, *type*, with name, `name`, from the IVI configuration store object, `obj`. *type* can be `HardwareAsset`, `DriverSession`, or `LogicalName`. If an entry of type, *type*, with name, `name`, does not exist, an error will occur.

`remove(obj, struct)` removes an entry using the fields in `struct`. If an entry with the *type* and name field in `struct` does not exist, an error will occur.

The modified configuration store object, `obj`, can be saved to the configuration store data file with the `commit` function.

If you attempt to remove an entry that is actively referenced by another entry, an error will occur. For example, you cannot remove a hardware asset that is currently referenced by a driver session.

Examples

```
c = iviconfigurationstore;
remove(c, 'HardwareAsset', 'gpib1');
```

See Also

iviconfigurationstore | add | commit | update

resolvehost

Network name or network address

Syntax

```
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.yourdomain.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.

```
[name,address] = resolvehost('144.212.100.10')
```

```
name = resolvehost('144.212.100.10','name')
address = resolvehost('www.yourdomain.com','address')
```

See Also
tcpip | udp

save

Save instrument objects and variables to MAT-file

Syntax

```
save filename
save filename obj1 obj2 ...
```

Arguments

`filename`	The MAT-file name.
`obj1 obj2 ...`	Instrument objects or arrays of instrument objects.

Description

`save filename` saves all MATLAB variables to the MAT-file `filename`. If an extension is not specified for `filename`, then a .mat extension is used.

`save filename obj1 obj2 ...` saves the instrument objects `obj1 obj2` ... to the MAT-file `filename`.

Examples

This example illustrates how to use the command form and the functional form of `save`.

```
s = serial('COM1');
set(s,'BaudRate',2400,'StopBits',1)
save MySerial1 s
set(s,'BytesAvailableFcn',@mycallback)
save('MySerial2','s')
```

More About

Tips

You can use `save` in the functional form as well as the command form shown above. When using the functional form, you must specify the filename and instrument objects as strings. For example, on a Windows machine, save the serial port object `s` to the file `MySerial.mat`,

```
s = serial('COM1');
save('MySerial','s')
```

Any data that is associated with the instrument object is not automatically stored in the MAT-file. For example, suppose there is data in the input buffer for `obj`. To save that data to a MAT-file, you must bring the data into the MATLAB workspace using one of the synchronous read functions, and then save the data to the MAT-file using a separate variable name. You can also save data to a text file with the `record` function.

You return objects and variables to the MATLAB workspace with the `load` command. Values for read-only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. To determine if a property is read-only, examine its reference pages or use the `propinfo` function.

See Also
`instrhelp` | `load` | `propinfo` | `record` | `Status`

scanstr

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 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.

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'
```

More About

Tips

Before you can read data from the instrument, it must be connected to obj with the fopen function. A connected interface object has a Status property value of open. An

error is returned if you attempt to perform a read operation while `obj` is not connected to the instrument.

If `msg` is not included as an output argument and the read operation was not successful, then a warning message is returned to the command line.

The `ValuesReceived` property value is increased by the number of values read — including the terminator — each time `scanstr` is issued.

See Also

`fopen` | `fscanf` | `instrhelp` | `sscanf` | `textread` | `EOSCharCode` | `EOSMode` | `Status` | `Terminator` | `ValuesReceived`

selftest

Run instrument self-test

Syntax

```
out = selftest(obj)
```

Arguments

obj	A device object.
out	The result of the self-test.

Description

`out = selftest(obj)` runs the self-test for the instrument associated with the device object specified by `obj`. The result of the self-test is returned to `out`. Note that the test result will vary based on the instrument.

serial

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.

Examples

This example creates the serial port object `s1` on a Windows machine associated with the serial port COM1.

```
s1 = serial('COM1');
```

The `Type`, `Name`, and `Port` properties are automatically configured.

```
s1.Type
ans =
    serial

s1.Name
ans =
    Serial-COM1

s1.Port
ans =
    COM
```

To specify properties during object creation,

```
s2 = serial('COM2','BaudRate',1200,'DataBits',7);
```

More About

Tips

At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with serial port objects.

```
instrhelp serial
```

When you create a serial port object, these property values are automatically configured:

- `Type` is given by `serial`.
- `Name` is given by concatenating `Serial` with the port specified in the `serial` function.
- `Port` is given by the port specified in the `serial` function.

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, the following commands are all valid.

```
s = serial('COM1','BaudRate',4800);
s = serial('COM1','baudrate',4800);
```

```
s = serial('COM1','BAUD',4800);
```

Before you can communicate with the instrument, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while `obj` is not connected to the instrument. You can connect only one serial port object to a given serial port.

See Also

`fclose` | `fopen` | `propinfo` | `Name` | `Port` | `Status` | `Type`

serialbreak

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.

More About

Tips

For some instruments, the break signal provides a way to clear the hardware buffer.

Before you can send a break to the instrument, it must be connected to `obj` with the `fopen` function. A connected serial port object has a `Status` property value of `open`. An error is returned if you attempt to send a break while `obj` is not connected to the instrument.

`serialbreak` is a synchronous function, and blocks the command line until execution is complete.

If you issue `serialbreak` while data is being asynchronously written, an error is returned. In this case, you must call the `stopasync` function or wait for the write operation to complete.

See Also
`fopen` | `stopasync` | `Status`

set

Configure or display instrument object properties

Syntax

```
set(obj)
props = set(obj)
set(obj,'PropertyName')
props = set(obj,'PropertyName')
set(obj,'PropertyName',PropertyValue,...)
set(obj,PN,PV)
set(obj,S)
```

Arguments

`obj`	An instrument object or an array of instrument objects.
`'PropertyName'`	A property name for `obj`.
`PropertyValue`	A property value supported by `PropertyName`.
`PN`	A cell array of property names.
`PV`	A cell array of property values.
`S`	A structure with property names and property values.
`props`	A structure array whose field names are the property names for `obj`, or cell array of possible values.

Description

`set(obj)` displays all configurable property values for `obj`. If a property has a finite list of possible 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.

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 ]
```

More About

Tips

You can use any combination of property name/property value pairs, structure arrays, and cell arrays in one call to set. Additionally, you can specify a property name without regard to case, and you can make use of property name completion. For example, if g is a GPIB object, then the following commands are all valid.

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

```
set(g,'eosmode')
set(g,'EOSM')
```

See Also

instrhelp | get | propinfo

size

Size of instrument object array

Syntax

```
d = size(obj)
[m,n] = size(obj)
[m1,m2,m3,...,mn] = size(obj)
m = size(obj,dim)
```

Arguments

obj	An instrument object or an array of instrument objects.
dim	The dimension of obj.
d	The number of rows and columns in obj.
m	The number of rows in obj, or the length of the dimension specified by dim.
n	The number of columns in obj.
m1,m2,...,mn	The length of the first N dimensions of obj.

Description

d = size(obj) returns the two-element row vector d containing the number of rows and columns in obj.

[m,n] = size(obj) returns the number of rows and columns in separate output variables.

[m1,m2,m3,...,mn] = size(obj) returns the length of the first n dimensions of obj.

m = size(obj,dim) returns the length of the dimension specified by the scalar dim. For example, size(obj,1) returns the number of rows.

See Also
instrhelp | length

spi

Create SPI object

Syntax

```
S = spi(Vendor,BoardIndex,Port);
```

Description

`S = spi(Vendor,BoardIndex,Port);` constructs an `spi` object associated with `Vendor`, `BoardIndex`, and `Port`. `Vendor` must be set to either `'aardvark'`, for use with a Total Phase Aardvark adaptor, or to `'ni845x'`, for use with the NI-845x adaptor board, to use this interface. `BoardIndex` specifies the board index of the hardware and is usually `0`. `Port` specifies the port number within the device and must be set to `0`.

SPI, or Serial Peripheral Interface, is a synchronous serial data link standard that operates in full duplex mode. Instrument Control Toolbox SPI support lets you open connections with individual chips and to read and write over the connections to individual chips using an Aardvark host adaptor.

The primary uses for the `spi` interface involve the `write`, `read`, and `writeAndRead` functions for synchronously reading and writing binary data. To identify SPI devices in the Instrument Control Toolbox, use the `instrhwinfo` function on the SPI interface, called `spi`.

Once the SPI object is created, there are properties that can be used to change communication settings. These includes properties for clock speed, clock phase, and clock polarity. For a list of all the properties and information about setting them, see the link for "Using Properties on the SPI Object" at the end of the Examples section.

Examples

Communicate With SPI Device

This example shows how to create a SPI object and communicate with a SPI device, using an Aardvark adaptor board.

Ensure that the Aardvark adaptor is installed so that you can use the `spi` interface, and then look at the adaptor properties.

```
instrhwinfo('spi')
instrhwinfo('spi', 'aardvark')

ans =

          VendorName: 'aardvark'
   VendorDescription: 'Total Phase I2C/SPI Driver'
   VendorLibraryName: 'aardvark.dll'
   InstalledBoardIds: {[0]}
   BoardSerialNumbers: {'2237722838'}
   ObjectConstructors: {'spi('aardvark', 0, 0)'}
```

Construct a `spi` object called `S` using `Vendor` `'aardvark'`, with `BoardIndex` of `0`, and `Port` of `0`.

```
S = spi('aardvark', 0, 0);
```

You can optionally change property settings such as `BitRate`, `ClockPhase`, or `ClockPolarity`. For example, set the `ClockPhase` from the default of `FirstEdge`.

```
S.ClockPhase = 'SecondEdge'
```

For a list of all the properties and information about setting them, see the link for "Using Properties on the SPI Object" at the end of the Examples section.

Connect to the chip.

```
connect(S);
```

Read and write to the chip.

```
% Create a variable containing the data to write
dataToWrite = [3 0 0 0];

% Write the binary data to the chip
write(S, dataToWrite);

% Create a variable to contain 5 bytes of returned data
numData = 5

% Read the binary data from the chip
read(S, numData)
```

Disconnect the SPI device and clean up by clearing the object.

```
disconnect(S);
clear(S);
```

- "Using Properties on the SPI Object" on page 10-18

Input Arguments

`Vendor` — Adaptor board vendor
`'aardvark'` | `'ni845x'`

Adaptor board vendor, specified as the character string `'aardvark'` or `'ni845x'`. You need to use a Total Phase Aardvark adaptor or an NI-845x adaptor board to use the SPI interface. You must specify this as the first argument when you create the `spi` object.

Example: `S = spi('aardvark', 0, 0);`

Data Types: `char`

`BoardIndex` — Board index of your hardware
`0`

Board index of your hardware, specified as a numeric value. This is usually `0`. You must specify this as the second argument when you create the `spi` object.

Example: `S = spi('aardvark', 0, 0);`

Data Types: `double`

`Port` — Port number of your hardware
`0`

Port number of your hardware, specified as the number `0`. The Aardvark adaptor uses `0` as the port number. You must specify this as the third argument when you create the `spi` object.

Example: `S = spi('aardvark', 0, 0);`

Data Types: `double`

spoll

Perform serial poll on GPIB objects

Syntax

```
out = spoll(obj)
out = spoll(obj,val)
[out] = spoll(obj)
[out,statusByte] = spoll(obj)
[out] = spoll(obj,val)
[out,statusByte] = spoll(obj,val)
```

Arguments

`obj`	A GPIB object or an array of GPIB objects.
`val`	A numeric array containing the indices of the objects in `obj`, that must be ready for servicing before control is returned to the MATLAB Command Window.
`out`	The GPIB objects ready for servicing.
`statusByte`	The service request (SRQ) line status byte.

Description

`out = spoll(obj)` performs a serial poll on the instruments associated with `obj`. `out` contains the GPIB objects that are ready for servicing. If no objects are ready for servicing, then `out` is empty.

`out = spoll(obj,val)` performs a serial poll and waits until the instruments specified by `val` are ready for servicing. An error is returned if a value specified in `val` does not match an index value in `obj`.

Using this syntax, `spoll` blocks access to the MATLAB Command Window until the objects specified by `val` are ready for servicing, or a timeout occurs for each object specified by `val`. The timeout period is specified by the `Timeout` property.

[out] = spoll(obj) returns the object or an array of objects.

[out,statusByte] = spoll(obj) returns the status byte along with the object or an array of objects.

[out] = spoll(obj,val) returns the object and the value specified in the index value of the object.

[out,statusByte] = spoll(obj,val) returns the status byte along with the object and the value specified in the index value of the object.

Examples

If obj is a four-element array and val is set to [1 3], then spoll will block access to the MATLAB Command Window until the instruments connected to the first and third GPIB objects have both asserted their SRQ line, or a timeout occurs.

Example of second output argument:

```
g1 = gpib('ni', 0, 1);
g2 = gpib('ni', 0, 2);
fopen([g1 g2]);
out1 = spoll(g1);
out2 = spoll([g1 g2], 1);
out3 = spoll([g1 g2], [1 2])
[out4 statusBytes] = spoll([g1 g2])
[out5 statusBytes] = spoll([g1 g2], 2)
fclose([g1 g2]);
```

More About

Tips

Serial polling is a method of obtaining specific information from GPIB objects when they request service. When you perform a serial poll, out contains the GPIB object that has asserted its SRQ line.

If obj is an array of GPIB objects

- Each element of obj must have the same BoardIndex property value.

• Each element of `obj` is polled to determine if the instrument is ready for servicing.

If you specify a second output argument when you call an `spoll`, full serial poll bytes are returned in addition to the SRQ line status in the second argument.

See Also
gpib | spoll (visa) | BoardIndex | length | Timeout

spoll (visa)

Perform serial poll on VISA objects

Syntax

```
out = spoll(obj)
out = spoll(obj,val)
[out] = spoll(obj)
[out,statusByte] = spoll(obj)
[out] = spoll(obj,val)
[out,statusByte] = spoll(obj,val)
```

Arguments

`obj`	A VISA object or an array of VISA objects.
`val`	A numeric array containing the indices of the objects in `obj`, that must be ready for servicing before control is returned to the MATLAB Command Window.
`out`	The VISA objects ready for servicing.
`statusByte`	The service request (SRQ) line status byte.

Description

`out = spoll(obj)` performs a serial poll on the instruments associated with `obj`. `out` contains the VISA objects that are ready for servicing. If no objects are ready for servicing, then `out` is empty.

`out = spoll(obj,val)` performs a serial poll and waits until the instruments specified by `val` are ready for servicing. An error is returned if a value specified in `val` does not match an index value in `obj`.

Using this syntax, `spoll` blocks access to the MATLAB Command Window until the objects specified by `val` are ready for servicing, or a timeout occurs for each object specified by `val`. The timeout period is specified by the `Timeout` property.

[out] = spoll(obj) returns the object or an array of objects.

[out,statusByte] = spoll(obj) returns the status byte along with the object or an array of objects.

[out] = spoll(obj,val) returns the object and the value specified in the index value of the object.

[out,statusByte] = spoll(obj,val) returns the status byte along with the object and the value specified in the index value of the object.

Examples

If obj is a four-element array and val is set to [1 3], then spoll will block access to the MATLAB Command Window until the instruments connected to the first and third VISA objects have both requested servicing, or a timeout occurs.

Example of second output argument:

```
v1 = visa('agilent', 'TCPIP0::yourdomainname.com::inst0::INSTR');
v2 = visa('agilent', 'TCPIP0::yourdomainname.com::inst01::INSTR');
fopen([v1 v2]);
out1 = spoll(v1);
out2 = spoll([v1 v2], 1);
out3 = spoll([v1 v2], [1 2])
[out4 statusBytes] = spoll([v1 v2])
[out5 statusBytes] = spoll([v1 v2], 2)
fclose([v1 v2]);
```

More About

Tips

Serial polling is a method of obtaining specific information from VISA objects when they request service. When you perform a serial poll, out contains the VISA object that have requested servicing.

If obj is an array of VISA objects

- Each element of obj must have the same BoardIndex property value.
- Each element of obj is polled to determine if the instrument is ready for servicing.

If you specify a second output argument when you call an `spoll`, full serial poll bytes are returned in addition to the SRQ line status in the second argument.

See Also
`visa` | `spoll` | `BoardIndex` | `Timeout`

stopasync

Stop asynchronous read and write operations

Syntax

```
stopasync(obj)
```

Arguments

obj	An interface object or an array of interface objects.

Description

stopasync(obj) stops any asynchronous read or write operation that is in progress for obj.

More About

Tips

You can write data asynchronously using the fprintf or fwrite functions. You can read data asynchronously using the readasync function, or by configuring the ReadAsyncMode property to continuous (serial port, TCPIP, UDP, and VISA-serial objects). In-progress asynchronous operations are indicated by the TransferStatus property.

If obj is an array of interface objects and one of the objects cannot be stopped, the remaining objects in the array are stopped and a warning is returned. After an object stops,

- Its TransferStatus property is configured to idle.
- Its ReadAsyncMode property is configured to manual (serial port, TCPIP, UDP, and VISA-serial objects).

• The data in its output buffer is flushed.

Data in the input buffer is not flushed. You can return this data to the MATLAB workspace using any of the synchronous read functions. If you execute the `readasync` function, or configure the `ReadAsyncMode` property to `continuous`, then the new data is appended to the existing data in the input buffer.

See Also

`fprintf` | `fwrite` | `readasync` | `ReadAsyncMode` | `TransferStatus`

tcpip

Create TCPIP object

Syntax

```
obj = tcpip('rhost')
obj = tcpip('rhost',rport)
obj = tcpip(...,'PropertyName',PropertyValue,...)
obj = tcpip('localhost', 30000, 'NetworkRole', 'client')
```

Arguments

`'rhost'`	The remote host.
`rport`	The remote port.
`'NetworkRole'`	Enables support for Server Sockets, using two values, `client` or `server`, to establish a connection as the client or the server.
`'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.

`obj = tcpip('localhost', 30000, 'NetworkRole', 'client')` creates a TCPIP object, `obj`, that is a client interface for a server socket.

Examples

Start a TCP/IP echo server and create a TCPIP object.

```
echotcpip('on',4012)
t = tcpip('localhost',4012);
```

Connect the TCPIP object to the host.

```
fopen(t)
```

Write to the host and read from the host.

```
fwrite(t,65:74)
A = fread(t, 10);
```

Disconnect the TCPIP object from the host and stop the echo server.

```
fclose(t)
echotcpip('off')
```

More About

Tips

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.

```
t = tcpip('144.212.113.252','InputBufferSize',1024);
t = tcpip('144.212.113.252','inputbuffersize',1024);
t = tcpip('144.212.113.252','INPUT',1024);
```

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.

- "Using TCP/IP Server Sockets" on page 7-52

See Also

`fopen` | `udp` | `sendmail` | `urlread` | `urlwrite` | `LocalHost` | `LocalPort` | `LocalPortMode` | `Name` | `RemoteHost` | `RemotePort` | `Status` | `Type` | `NetworkRole`

tmtool

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 a file

For a full description of the Test & Measurement Tool with examples, see "Test & Measurement Tool Overview" on page 17-2.

See Also
`midedit` | `midtest`

trigger

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`.

More About

Tips

Before you can use `trigger`, `obj` must be connected to the instrument with the `fopen` function. A connected interface object has a `Status` property value of `open`. An error is returned if you attempt to use `trigger` while `obj` is not connected to the instrument.

For GPIB and VISA-GPIB objects, the Group Execute Trigger (GET) message is sent to the instrument.

For VISA-VXI objects, if the `TriggerType` property is configured to `software`, the Word Serial Trigger command is sent to the instrument. If `TriggerType` is configured to `hardware`, a hardware trigger is sent on the line specified by the `TriggerLine` property.

See Also
fopen | Status | TriggerLine | TriggerType

udp

Create UDP object

Syntax

```
obj = udp('')
obj = udp('rhost')
obj = udp('rhost',rport)
obj = udp(...,'PropertyName',PropertyValue,...)
```

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

`obj = udp('')` creates a UDP object, `obj`, not associated with a remote host. `obj = udp('rhost')` creates a UDP object associated with remote host `rhost`.

`obj = udp('rhost',rport)` creates a UDP object with remote port value, `rport`. The default remote port is 9090.

`obj = udp(...,'PropertyName',PropertyValue,...)` 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.

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)
```

More About

Tips

At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with UDP objects.

```
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.

See Also

`fopen` | `LocalHost` | `LocalPort` | `LocalPortMode` | `Name` | `RemoteHost` | `RemotePort` | `Status` | `Type`

update

Update entry of IVI configuration store object

Syntax

```
update(obj, 'type', 'name', 'P1', V1, ...)
update(obj, struct)
```

Arguments

`obj`	IVI configuration store object.
`'type'`	Type of entry; *type* can be `HardwareAsset`, `DriverSession`, or `LogicalName`.
`'name'`	Name of the `DriverSession`, `HardwareAsset`, or `LogicalName` to be updated.
`'P1'`	First parameter for updated entry; other parameter-value pairs may follow.
`V1`	Value for first parameter.
`struct`	Structure defining entry fields to be updated.

Description

`update(obj, 'type', 'name', 'P1', V1, ...)` updates an entry of type, *type*, with name, `name`, in IVI configuration store object, `obj`, using the specified parameter-value pairs. *type* can be `HardwareAsset`, `DriverSession`, or `LogicalName`.

If an entry of type, *type* with name, `name` does not exist, an error will occur.

Valid parameters for a `DriverSession` are listed below. The default value for on/off parameters is `off`.

Parameter	Value	Description
Name	string	A unique name for the driver session.

Parameter	Value	Description
SoftwareModule	string	The name of a software module entry in the configuration store.
HardwareAsset	string	The name of a hardware asset entry in the configuration store.
Description	Any string	Description of driver session
VirtualNames	structure	A struct array containing virtual name mappings
Cache	on/off	Enable caching if the driver supports it.
DriverSetup	Any string	This value is software module dependent
InterchangeCheck	on/off	Enable driver interchangeability checking, if supported
QueryInstrStatus	on/off	Enable instrument status querying by the driver
RangeCheck	on/off	Enable extended range checking by the driver, if supported
RecordCoercions	on/off	Enable recording of coercions by the driver, if supported
Simulate	on/off	Enable simulation by the driver

Valid fields for `HardwareAsset` are

Parameter	Value	Description
Name	string	A unique name for the hardware asset
Description	Any string	Description of hardware asset
IOResourceDescriptor	string	The I/O address of the hardware asset

Valid fields for `LogicalName` are

Parameter	Value	Description
Name	string	A unique name for the logical name
Description	Any string	Description of hardware asset
Session	string	The name of a driver session entry in the configuration store

update(obj, struct) updates the entry using the fields in struct. If an entry with the type and name field in struct does not exist, an error will occur. Note that the name field cannot be updated using this syntax.

Examples

Update the Description parameter of the driver session named ScopeSession in the IVI configuration store object named c.

```
c = iviconfigurationstore;
update(c, 'DriverSession', 'ScopeSession', 'Description', ...
'A session.');
```

See Also

iviconfigurationstore | add | commit | remove

visa

Create VISA object

Syntax

```
obj = visa('vendor','rsrcname')
```

Arguments

`'vendor'`	A supported VISA vendor.
`'rsrcname'`	The resource name of the VISA instrument.
`'PropertyName'`	A VISA property name.
`PropertyValue`	A property value supported by `PropertyName`.
`obj`	The VISA object.

Description

`obj = visa('vendor','rsrcname')` creates the VISA object `obj` with a resource name given by `rsrcname` for the vendor specified by `vendor`.

You must first configure your VISA resources in the vendor's tool first, and then you create these VISA objects. Use `instrhwinfo` to find the commands to configure the objects:

```
vinfo = instrhwinfo('visa','agilent');
vinfo.ObjectConstructorName
```

If an invalid vendor or resource name is specified, an error is returned and the VISA object is not created. For a list of supported vales for `vendor` see Supported Vendor and Resource Names.

Examples

Create a VISA-serial object connected to serial port COM1 using National Instruments VISA interface.

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

Create a VISA-GPIB object connected to board 0 with primary address 1 and secondary address 30 using Agilent Technologies VISA interface.

```
vg = 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 = 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 = visa('agilent','GPIB-VXI1::72::INSTR');
```

Create a VISA-RSIB object connected to an instrument configured with IP address 192.168.1.33.

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

Create a VISA-TCPIP object connected to an instrument configured with IP address 216.148.60.170.

```
vt = visa('tek', 'TCPIP::216.148.60.170::INSTR')
```

Create a VISA-USB object connected to a USB instrument with manufacturer ID 0x1234, model code 125, and serial number A22-5.

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

More About

Tips

At any time, you can use the `instrhelp` function to view a complete listing of properties and functions associated with VISA objects.

```
instrhelp visa
```

You can specify the property names and property values using any format supported by the `set` function. For example, you can use property name/property value cell array pairs. Additionally, you can specify property names without regard to case, and you can make use of property name completion. For example, the following commands are all valid.

```
v = visa('ni','GPIB0::1::INSTR','SecondaryAddress', 96);
v = visa('ni','GPIB0::1::INSTR','secondaryaddress', 96);
v = visa('ni','GPIB0::1::INSTR','SECOND', 96);
```

Before you can communicate with the instrument, it must be connected to `obj` with the `fopen` function. A connected VISA object has a `Status` property value of `open`. An error is returned if you attempt a read or write operation while `obj` is not connected to the instrument. You cannot connect multiple VISA objects to the same instrument.

Creating a VISA-GPIB Object

When you create a VISA-GPIB object, these properties are automatically configured:

- `Type` is given by `visa-gpib`.
- `Name` is given by concatenating `VISA-GPIB` with the board index, the primary address, and the secondary address.
- `BoardIndex`, `PrimaryAddress`, `SecondaryAddress`, and `RsrcName` are given by the values specified during object creation.

Creating a VISA-GPIB-VXI Object

When you create a VISA-GPIB-VXI object, these properties are automatically configured:

- `Type` is given by `visa-gpib-vxi`.
- `Name` is given by concatenating `VISA-GPIB-VXI` with the chassis index and the logical address specified in the `visa` function.
- `ChassisIndex`, `LogicalAddress`, and `RsrcName` are given by the values specified during object creation.
- `BoardIndex`, `PrimaryAddress`, and `SecondaryAddress` are given by the `visa` driver after the object is connected to the instrument with `fopen`.

Creating a VISA-RSIB Object

When you create a VISA-RSIB object, these properties are automatically configured:

- `Type` is given by `visa-rsib`.
- `Name` is given by concatenating `VISA-RSIB` with the remote host specified in the `visa` function.
- `RemoteHost` and `RsrcName` are given by the values specified during object creation.

Creating a VISA-Serial Object

When you create a VISA-serial object, these properties are automatically configured:

- `Type` is given by `visa-serial`.
- `Name` is given by concatenating `VISA-Serial` with the port specified in the `visa` function.
- `Port` and `RsrcName` are given by the values specified during object creation.

Creating a VISA-TCPIP Object

When you create a VISA-TCPIP object, these properties are automatically configured:

- `Type` is given by `visa-tcpip`.
- `Name` is given by concatenating `VISA-TCPIP` with the board index, remote host, and LAN device name specified in the `visa` function.
- `BoardIndex`, `RemoteHost`, `LANNAme`, and `RsrcName` are given by the values specified during object creation.

Creating a VISA-USB Object

When you create a VISA-USB object, these properties are automatically configured:

- `Type` is given by `visa-usb`.
- `Name` is given by concatenating `VISA-USB` with the board index, manufacturer ID, model code, serial number, and interface number specified in the `visa` function.
- `BoardIndex`, `ManufacturerID`, `ModelCode`, `SerialNumber`, `InterfaceIndex`, and `RsrcName` are given by the values specified during object creation.

Creating a VISA-VXI Object

When you create a VISA-VXI object, these properties are automatically configured:

- `Type` is given by `visa-vxi`.
- `Name` is given by concatenating `VISA-VXI` with the chassis index and the logical address specified in the `visa` function.
- `ChassisIndex`, `LogicalAddress`, and `RsrcName` are given by the values specified during object creation.

See Also

`fclose` | `fopen` | `instrhelp` | `instrhwinfo` | `BoardIndex` | `ChassisIndex` | `InterfaceIndex` | `LANName` | `LogicalAddress` | `ManufacturerID` | `ModelCode` | `Name` | `Port` | `PrimaryAddress` | `RsrcName` | `SecondaryAddress` | `SerialNumber` | `Status` | `Type`

write

Write binary data to SPI instrument

Syntax

```
write(OBJ, A)
```

Description

`write(OBJ, A)` writes the data, A, to the SPI instrument connected to interface object, OBJ. OBJ must be a 1-by-1 SPI interface object. By default the `'uint8'` precision is used.

The interface object must be connected to the device with the `connect` function before any data can be read from or written to the device, otherwise an error is returned. A connected interface object has a `ConnectionStatus` property value of connected.

The SPI protocol operates in full duplex mode, input and output data transfers happen simultaneously. For every byte written to the device, a byte is read back from the device. This function will automatically flush the incoming data.

Examples

This example shows how to create a SPI object `s`, and read and write data.

Construct a `spi` object called `s` using `Vendor` `'aardvark'`, with `BoardIndex` of `0`, and `Port` of `0`.

```
s = spi('aardvark', 0, 0);
```

Connect to the chip.

```
connect(s);
```

Write to the chip.

```
dataToWrite = [2 0 0 255]
```

```
write(s, dataToWrite);
```

Disconnect the SPI device and clean up by clearing the object.

```
disconnect(s);
clear('s');
```

writeAndRead

Write and read binary data from SPI instrument

Syntax

```
A = writeAndRead(OBJ, dataToWrite)
```

Description

`A = writeAndRead(OBJ, dataToWrite)` writes the data, `dataToWrite`, to the instrument connected to interface object OBJ and reads the data available from the instrument as a result of writing `dataToWrite`. OBJ must be a 1-by-1 SPI interface object. By default, `'uint8'` precision is used.

The interface object must be connected to the device using the `connect` function before any data can be read from the device, otherwise an error is returned. A connected interface object has a `ConnectionStatus` property value of `connected`.

SPI protocol operates in full duplex mode, so input and output data transfers happen simultaneously. For every byte written to the device, a byte is read back from the device.

For more information on using the SPI interface and this function, see "Configuring SPI Communication" on page 10-4 and "Transmitting Data Over the SPI Interface" on page 10-9.

Examples

This example shows how to create a SPI object `s`, and read and write data.

Construct a `spi` object called `s` using `Vendor 'aardvark'`, with `BoardIndex` of `0`, and `Port` of `0`.

```
s = spi('aardvark', 0, 0);
```

Connect to the chip.

```
connect(s);
```

Read and write to the chip.

```
dataToWrite = [2 0 0 255]
data = writeAndRead(s, dataToWrite);
```

Disconnect the SPI device and clean up by clearing the object.

```
disconnect(s);
clear('s');
```

Properties — Alphabetical List

ActualLocation

Configuration store file used by IVI configuration store object

Description

ActualLocation reflects the location of the IVI configuration store actually being used. It is either the master configuration store, or the ProcessLocation if an alternative to the master store was specified when the IVI configuration store object was created.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	String

Values

The default value is the master configuration store.

See Also

Functions

commit

Properties

MasterLocation, ProcessLocation

Alias

Alias of resource name for VISA instrument

Description

`Alias` indicates the alias for the resource name for a VISA instrument. When you create a VISA object, you can specify either the resource name for a VISA instrument or an alias for the resource name. If an alias is specified, `Alias` is automatically assigned the value specified in the VISA function. If a resource name is specified and the resource name has an alias, `Alias` is updated with the alias value. If the resource name does not have an alias, `Alias` is an empty 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 vender-supplied tools. You do not set an alias in the MATLAB workspace. When you create the VISA object in the MATLAB workspace, the `Alias` property of the object takes on the value of the resource name alias. You do not directly set the value of this property.

National Instruments' Measurement & Automation Explorer (MAX) is one example of a graphical interface tool for setting a VISA alias in NI-VISA. In this tool, select **Tools** > **NI-VISA** > **Alias Editor** to edit, add, or clear aliases. When you have your aliases defined, you can use them in the MATLAB workspace to access your resources.

See Also

Functions

visa

Properties

RsrcName

BaudRate

Specify bit transmit rate

Description

You configure `BaudRate` as bits per second. The transferred bits include the start bit, the data bits, the parity bit (if used), and the stop bits. However, only the data bits are stored.

The baud rate is the rate at which information is transferred in a communication channel. In the serial port context, "9600 baud" means that the serial port is capable of transferring a maximum of 9600 bits per second. If the information unit is one baud (one bit), then the bit rate and the baud rate are identical. If one baud is given as 10 bits, (for example, eight data bits plus two framing bits), the bit rate is still 9600 but the baud rate is 9600/10, or 960. You always configure `BaudRate` as bits per second. Therefore, in the above example, set `BaudRate` to 9600.

Note Both the computer and the instrument must be configured to the same baud rate before you can successfully read or write data.

Your system computes the acceptable rates by taking the baud base, which is determined by your serial port, and dividing it by a positive whole number divisor . The system will try to find the best match by modifying the divisor. For example, if:

```
baud base = 115200 bits per second
divisors = 1,2,3,4,5….
Possible BaudRates = 115200, 57600, 38400, 28800, 23040….
```

Your system may further limit the available baud rates to conform to specific conventions or standards. In the above example, for instance, 23040 bits/sec may not be available on all systems.

Characteristics

Usage	Serial port, VISA-serial

Read only	Never
Data type	Double

Values

The default value is 9600.

See Also

Properties

DataBits, Parity, StopBits

BoardIndex

Specify index number of interface board

Description

You configure `BoardIndex` to be the index number of the GPIB board, USB board, or TCP/IP board associated with your instrument. When you create a GPIB, VISA-GPIB, VISA-GPIB-VXI, VISA-TCPIP, or VISA-USB object, `BoardIndex` is automatically assigned the value specified in the `gpib` or `visa` function.

For GPIB objects, the `Name` property is automatically updated to reflect the `BoardIndex` value. For VISA-GPIB, VISA-GPIB-VXI, VISA-TCPIP, or VISA-USB objects, the `Name` and `RsrcName` properties are automatically updated to reflect the `BoardIndex` value.

You can configure `BoardIndex` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-GPIB-VXI, VISA-TCPIP, VISA-USB
Read only	While open (GPIB, VISA-GPIB), always (VISA-GPIB-VXI, VISA-TCPIP, VISA-USB)
Data type	Double

Values

The value is defined after the instrument object is created.

Examples

Suppose you create a VISA-GPIB object associated with board 4, primary address 1, and secondary address 8.

```
vg = visa('agilent','GPIB4::1::8::INSTR');
```

The BoardIndex, Name, and RsrcName properties reflect the GPIB board index number.

```
vg.BoardIndex
ans =
    [4]

vg.Name
ans =
    'VISA-GPIB4-1-8'

vg,.RsrcName
ans =
  'GPIB4::1::8::INSTR'
```

See Also

Functions

fclose, gpib, visa

Properties

Name, RsrcName, Status

BreakInterruptFcn

Specify callback function to execute when break-interrupt event occurs

Description

You configure `BreakInterruptFcn` to execute a callback function when a break-interrupt event occurs. A break-interrupt event is generated by the serial port when the received data is in an off (space) state longer than the transmission time for one byte.

Note A break-interrupt event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a break-interrupt event occurs, the record file records this information:

- The event type as `BreakInterrupt`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

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`

BusManagementStatus

State of GPIB bus management lines

Description

BusManagementStatus is a structure array that contains the fields Attention, InterfaceClear, RemoteEnable, ServiceRequest, and EndOrIdentify. These fields indicate the state of the Attention (ATN), Interface Clear (IFC), Remote Enable (REN), Service Request (SRQ) and End Or Identify (EOI) GPIB lines.

BusManagementStatus can be on or off for any of these fields. If BusManagementStatus is on, the associated line is asserted. If BusManagementStatus is off, the associated line is unasserted.

Characteristics

Usage	GPIB
Read only	Always
Data type	Structure

Values

off	The GPIB line is unasserted
on	The GPIB line is asserted

The default value is instrument dependent.

Examples

Create the GPIB object g associated with a National Instruments board, and connect g to a Tektronix TDS 210 oscilloscope.

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

```
fopen(g)
```

Write the `*STB?` command, which queries the instrument's status byte register, and then return the state of the bus management lines with the `BusManagementStatus` property.

```
fprintf(g,'*STB?')
g.BusManagementStatus
ans =
         Attention: 'off'
    InterfaceClear: 'off'
      RemoteEnable: 'on'
    ServiceRequest: 'off'
     EndOrIdentify: 'on'
```

REN is asserted because the system controller placed the scope in the remote enable mode, while EOI is asserted to mark the end of the command.

Now read the result of the `*STB?` command, and return the state of the bus management lines.

```
out = fscanf(g)
out =
O
g.busmanagementstatus
ans =
         Attention: 'off'
    InterfaceClear: 'off'
      RemoteEnable: 'on'
    ServiceRequest: 'off'
     EndOrIdentify: 'off'

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

ByteOrder

Specify byte order of instrument

Description

You configure `ByteOrder` to be `littleEndian` or `bigEndian`. If `ByteOrder` is `littleEndian`, then the instrument stores the first byte in the first memory address. If `ByteOrder` is `bigEndian`, then the instrument stores the last byte in the first memory address.

For example, suppose the hexadecimal value 4F52 is to be stored in instrument memory. Because this value consists of two bytes, 4F and 52, two memory locations are used. Using big-endian format, 4F is stored first in the lower storage address. Using little-endian format, 52 is stored first in the lower storage address.

Note You should configure `ByteOrder` to the appropriate value for your instrument before performing a read or write operation. Refer to your instrument documentation for information about the order in which it stores bytes.

You can set this property on interface objects such as TCP/IP or GPIB. In this example, a TCP/IP object, `Tobj`, is set to `bigEndian` and you change it to `littleEndian`.

```
Tobj.ByteOrder = 'littleEndian'
```

Characteristics

Usage	Any instrument object
Read only	Never
Data type	String

Values

{littleEndian}	The byte order of the instrument is little-endian.

bigEndian   The byte order of the instrument is big-endian.

See Also

Properties

Status

BytesAvailable

Number of bytes available in input buffer

Description

`BytesAvailable` indicates the number of bytes currently available to be read from the input buffer. The property value is continuously updated as the input buffer is filled, and is set to `0` after the `fopen` function is issued.

You can make use of `BytesAvailable` only when reading data asynchronously. This is because when reading data synchronously, control is returned to the MATLAB Command Window only after the input buffer is empty. Therefore, the `BytesAvailable` value is always 0. To learn how to read data asynchronously, refer to "Synchronous Versus Asynchronous Read Operations" on page 3-21.

The `BytesAvailable` value can range from zero to the size of the input buffer. Use the `InputBufferSize` property to specify the size of the input buffer. Use the `ValuesReceived` property to return the total number of values read.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Double

Values

The value can range from zero to the size of the input buffer. The default value is `0`.

See Also

Functions

`fopen`

Properties

`InputBufferSize`, `TransferStatus`, `ValuesReceived`

BytesAvailableFcn

Specify callback function to execute when specified number of bytes are available in input buffer, or terminator is read

Description

You configure `BytesAvailableFcn` to execute a callback function when a bytes-available event occurs. A bytes-available event occurs when the number of bytes specified by the `BytesAvailableFcnCount` property is available in the input buffer, or after a terminator is read, as determined by the `BytesAvailableFcnMode` property.

Note A bytes-available event can be generated only for asynchronous read operations.

If the `RecordStatus` property value is `on`, and a bytes-available event occurs, the record file records this information:

- The event type as `BytesAvailable`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Note: You cannot use ASCII values larger than 127 characters. The function is limited to 127 binary characters.

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.

Examples

Create the serial port object s on a Windows machine for a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1.

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

Configure s to execute the callback function instrcallback when 40 bytes are available in the input buffer.

```
s.BytesAvailableFcnCount = 40;
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcn = @instrcallback;
```

Connect s to the oscilloscope.

```
fopen(s)
```

Write the *IDN? command, which instructs the scope to return identification information. Because the default value for the ReadAsyncMode property is continuous, data is read as soon as it is available from the instrument.

```
fprintf(s,'*IDN?')
```

The resulting output from instrcallback is shown below.

```
BytesAvailable event occurred at 18:33:35 for the object:
Serial-COM1.
```

56 bytes are read and instrcallback is called once. The resulting display is shown above.

```
s.BytesAvailable
ans =
    56
```

Suppose you remove 25 bytes from the input buffer and issue the MEASUREMENT? command, which instructs the scope to return its measurement settings.

```
out = fscanf(s,'%c',25);
fprintf(s,'MEASUREMENT?')
```

The resulting output from `instrcallback` is shown below.

```
BytesAvailable event occurred at 18:33:48 for the object:
Serial-COM1.

BytesAvailable event occurred at 18:33:48 for the object:
Serial-COM1.
```

There are now 102 bytes in the input buffer, 31 of which are left over from the `*IDN?` command. `instrcallback` is called twice; once when 40 bytes are available and once when 80 bytes are available.

```
s.BytesAvailable
ans =
    102
```

See Also

Functions

`record`

Properties

`BytesAvailableFcnCount`, `BytesAvailableFcnMode`, `EOSCharCode`, `RecordStatus`, `Terminator`, `TransferStatus`

BytesAvailableFcnCount

Specify number of bytes that must be available in input buffer to generate bytes-available event

Description

You configure `BytesAvailableFcnCount` to the number of bytes that must be available in the input buffer before a bytes-available event is generated.

Use the `BytesAvailableFcnMode` property to specify whether the bytes-available event occurs after a certain number of bytes are available or after a terminator is read.

The bytes-available event executes the callback function specified for the `BytesAvailableFcn` property.

You can configure `BytesAvailableFcnCount` only when the object is disconnected from the instrument. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	Any instrument object
Read only	While open
Data type	Double

Values

The default value is `48`.

See Also

Functions

`fclose`

Properties

BytesAvailableFcn, BytesAvailableFcnMode, EOSCharCode, Status, Terminator

BytesAvailableFcnMode

Specify whether bytes-available event is generated after specified number of bytes are available in input buffer, or after terminator is read

Description

For serial port, TCPIP, UDP, or VISA-serial objects, you can configure `BytesAvailableFcnMode` to be `terminator` or `byte`. For all other instrument objects, you can configure `BytesAvailableFcnMode` to be `eosCharCode` or `byte`.

If `BytesAvailableFcnMode` is `terminator`, a bytes-available event occurs when the terminator specified by the `Terminator` property is read. If `BytesAvailableFcnMode` is `eosCharCode`, a bytes-available event occurs when the End-Of-String character specified by the `EOSCharCode` property is read. If `BytesAvailableFcnMode` is `byte`, a bytes-available event occurs when the number of bytes specified by the `BytesAvailableFcnCount` property is available.

The bytes-available event executes the callback function specified for the `BytesAvailableFcn` property.

You can configure `BytesAvailableFcnMode` only when the object is disconnected from the instrument. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	Any instrument object
Read only	While open
Data type	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

Number of bytes currently in output buffer

Description

`BytesToOutput` indicates the number of bytes currently in the output buffer waiting to be written to the instrument. The property value is continuously updated as the output buffer is filled and emptied, and is set to `0` after the `fopen` function is issued.

You can make use of `BytesToOutput` only when writing data asynchronously. This is because when writing data synchronously, control is returned to the MATLAB Command Window only after the output buffer is empty. Therefore, the `BytesToOutput` value is always `0`. To learn how to write data asynchronously, Refer to "Synchronous Versus Asynchronous Write Operations" on page 3-16.

Use the `ValuesSent` property to return the total number of values written to the instrument.

Note If you attempt to write out more data than can fit in the output buffer, then an error is returned and `BytesToOutput` is `0`. You specify the size of the output buffer with the `OutputBufferSize` property.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	Double

Values

The default value is `0`.

See Also

Functions

`fopen`

Properties

`OutputBufferSize`, `TransferStatus`, `ValuesSent`

ChassisIndex

Specify index number of VXI chassis

Description

You configure `ChassisIndex` to be the index number of the VXI chassis associated with your instrument.

When you create a VISA-VXI or VISA-GPIB-VXI object, `ChassisIndex` is automatically assigned the value specified in the `visa` function. For both object types, the `Name` and `RsrcName` properties are automatically updated to reflect the `ChassisIndex` value.

You can configure `ChassisIndex` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	While open
Data type	double

Values

The value is defined after the instrument object is created.

Examples

Suppose you create a VISA-GPIB-VXI object associated with chassis 0 and logical address 32.

```
v = visa('agilent','GPIB-VXI0::32::INSTR');
```

The ChassisIndex, Name, and RsrcName properties reflect the VXI chassis index number.

```
v.ChassisIndex
ans =
    [0]

v.Name
ans =
    'VISA-GPIB-VXI0-32'

v.RsrcName
ans =
    'GPIB-VXI0::32::INSTR'
```

See Also

Functions

fclose, visa

Properties

Name, RsrcName, Status

CompareBits

Specify number of bits that must match EOS character to complete read operation, or to assert EOI line

Description

You can configure `CompareBits` to be `7` or `8`. If `CompareBits` is `7`, the read operation completes when a byte that matches the low seven bits of the End-Of-String (EOS) character is received. The End Or Identify (EOI) line is asserted when a byte that matches the low seven bits of the EOS character is written. If `CompareBits` is `8`, the read operation completes when a byte that matches all eight bits of the EOS character is received. The EOI line is asserted when a byte that matches all eight bits of the EOS character is written.

You can specify the EOS character with the `EOSCharCode` property. You can specify when the EOS character is used (read operation, write operation, or both) with the `EOSMode` property.

Characteristics

Usage	GPIB
Read only	Never
Data type	Double

Values

{8}	Compare all eight EOS bits.
7	Compare the lower seven EOS bits.

See Also

Properties

`EOSCharCode, EOSMode`

ConfirmationFcn

Specify callback function to execute when confirmation event occurs

Description

You configure `ConfirmationFcn` to execute a callback function when a confirmation event occurs.

A confirmation event is generated when the command written to the instrument results in the instrument being configured to a different value than it was sent.

Note A confirmation event can be generated only when the object is connected to the instrument with `connect`.

Characteristics

Usage	Device
Read only	Never
Data type	Callback

Values

The default value is an empty string.

See Also

Functions

connect

DataBits

Specify number of data bits to transmit

Description

You can configure `DataBits` to be 5, 6, 7, or 8. Data is transmitted as a series of five, six, seven, or eight bits with the least significant bit sent first. At least seven data bits are required to transmit ASCII characters. Eight bits are required to transmit binary data. Five and six bit data formats are used for specialized communication equipment.

Note Both the computer and the instrument must be configured to transmit the same number of data bits.

In addition to the data bits, the serial data format consists of a start bit, one or two stop bits, and possibly a parity bit. You specify the number of stop bits with the `StopBits` property, and the type of parity checking with the `Parity` property.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	Double

Values

`DataBits` can be 5, 6, 7, or 8. The default value is 8.

See Also

Properties

`Parity`, `StopBits`

DatagramAddress

IP dotted decimal address of received datagram sender

Description

`DatagramAddress` is the datagram sender IP address of the next datagram to be read from the input buffer. An example of an IP dotted decimal address 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`

DatagramPort

Port number of datagram sender

Description

DatagramPort is the port number of the next datagram to be read from the input buffer. When you read a datagram from the input buffer, DatagramPort is updated.

Characteristics

Usage	UDP
Read only	Never
Data type	Double

Values

The default value is [ ].

See Also

Functions

udp

Properties

DatagramAddress

DatagramReceivedFcn

Specify callback function to execute when datagram is received

Description

You configure `DatagramReceivedFcn` to execute a callback function when a datagram has been received. The callback executes when a complete datagram is received in the input buffer.

Note A datagram-received event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a datagram-received event occurs, the record file records this information:

- The event type as `DatagramReceived`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

To learn how to create a callback function, refer to "Creating and Executing Callback Functions" on page 4-32

Characteristics

Usage	UDP
Read only	Never
Data type	Callback

Values

The default value is `''`.

See Also

Functions

`readasync`, `udp`

Properties

`DatagramAddress`, `DatagramPort`, `ReadAsyncMode`

DatagramTerminateMode

Configure terminate read mode when reading datagrams

Description

`DatagramTerminateMode` defines how `fread` and `fscanf` read operations terminate. You can configure `DatagramTerminateMode` to be `on` or `off`.

If `DatagramTerminateMode` is `on`, the read operation terminates when a datagram is read. When `DatagramTerminateMode` is `off`, `fread` and `fscanf` read across datagram boundaries.

Characteristics

Usage	UDP
Read only	Never
Data type	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

Specify state of DTR pin

Description

You can configure `DataTerminalReady` to be `on` or `off`. If `DataTerminalReady` is `on`, the Data Terminal Ready (DTR) pin is asserted. If `DataTerminalReady` is `off`, the DTR pin is unasserted.

In normal usage, the DTR and Data Set Ready (DSR) pins work together, and are used to signal if instruments are connected and powered. However, there is nothing in the RS-232 or the RS-485 standard that states the DTR pin must be used in any specific way. For example, DTR and DSR might be used for handshaking. You should refer to your instrument documentation to determine its specific pin behavior.

You can return the value of the DSR pin with the `PinStatus` property. Handshaking is described in "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`

DriverName

Specify name of driver used to communicate with instrument

Description

For device objects with a `DriverType` property value of `MATLAB Instrument Driver`, the `DriverName` property specifies the name of the MATLAB instrument driver that contains the supported instrument commands.

For device objects with a `DriverType` property value of `VXIplug&play`, `IVI-C`, or `IVI-COM`, the `DriverName` is the name of the VXI*plug&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

Array of driver sessions contained in IVI configuration store

Description

`DriverSessions` identifies all the driver sessions in the IVI configuration store. Each driver session maps a software module to a hardware asset and its `IOResourceDescriptor`. A driver session also determines default settings and behavior for its software module.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

See Also

Properties

`HardwareAssets`, `SoftwareModules`

DriverType

Specify type of driver used to communicate with instrument

Description

`DriverType` can be `MATLAB interface object`, `MATLAB VXIplug&play`, or `MATLAB IVI`. If `DriverType` is `MATLAB interface object`, an interface object is used to communicate with the instrument. If `DriverType` is `MATLAB VXIplug&play`, a VXI*plug&play* driver is used to communicate with the instrument. If `DriverType` is `MATLAB IVI`, an IVI driver is used to communicate with the instrument.

Characteristics

Usage	Device
Read only	Always
Data type	String

Values

The `DriverType` value is defined at the device object creation. `DriverType` can be `MATLAB interface object`, `MATLAB VXIplug&play`, or `MATLAB IVI`.

See Also

Properties

`DriverName`

EOIMode

Specify if EOI line is asserted at end of write operation

Description

You can configure `EOIMode` to be `on` or `off`. If `EOIMode` is `on`, the End Or Identify (EOI) line is asserted at the end of a write operation. If `EOIMode` is `off`, the EOI line is not asserted at the end of a write operation. `EOIMode` applies to both binary and text write operations.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI
Read only	Never
Data type	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

EOSCharCode

Specify EOS character

Description

You can configure `EOSCharCode` to an integer value ranging from `0` to `255`, or to the equivalent ASCII character. For example, to configure `EOSCharCode` to a carriage return, you specify the value to be `CR` or `13`.

`EOSCharCode` replaces \n wherever it appears in the ASCII command sent to the instrument. Note that %s\n is the default format for the `fprintf` function.

For many practical applications, you will configure both `EOSCharCode` and the `EOSMode` property. `EOSMode` specifies when the EOS character is used. If `EOSMode` is `write` or `read&write` (writing is enabled), the EOI line is asserted every time the `EOSCharCode` value is written to the instrument. If `EOSMode` is `read` or `read&write` (reading is enabled), then the read operation might terminate when the `EOSCharCode` value is detected. For GPIB objects, the `CompareBits` property specifies the number of bits that must match the EOS character to complete a read or write operation.

To see how `EOSCharCode` and `EOSMode` work together, refer to the example given in the `EOSMode` property description.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-VXI, VISA-GPIB-VXI
Read only	Never
Data type	ASCII value

Values

An integer value ranging from `0` to `255` or the equivalent ASCII character. The default value is `LF`, which corresponds to a line feed.

See Also

Functions

`fprintf`

Properties

`CompareBits`, `EOSMode`

EOSMode

Specify when EOS character is written or read

Description

For GPIB, VISA-GPIB, VISA-VXI, 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 `ll see tEOSMode` 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.

```
g = gpib('ni',0,1);
fopen(g)
fprintf(g,'Volt?')
out = fscanf(g)
out =
+2.00000E+00
```

The EOSMode and EOSCharCode properties are configured to terminate the read operation when an E character is encountered.

```
g.EOSMode = 'read'
g.EOSCharCode = 'E'
fprintf(g,'Volt?')
out = fscanf(g)
out =
```

```
+2.00000
```

See Also

Properties

CompareBits, EOIMode, EOSCharCode

ErrorFcn

Specify callback function to execute when error event occurs

Description

You configure `ErrorFcn` to execute a callback function when an error event occurs.

Note An error event is generated only for asynchronous read and write operations.

An error event is generated when a timeout occurs. A timeout occurs if a read or write operation does not successfully complete within the time specified by the `Timeout` property. An error event is not generated for configuration errors such as setting an invalid property value.

If the `RecordStatus` property value is `on`, and an error event occurs, the record file records this information:

- The event type as `Error`
- The error message
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

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
```

FlowControl

Specify data flow control method to use

Description

You can configure `FlowControl` to be `none`, `hardware`, or `software`. If `FlowControl` is `none`, then data flow control (handshaking) is not used. If `FlowControl` is `hardware`, then hardware handshaking is used to control data flow. If `FlowControl` is `software`, then software handshaking is used to control data flow.

Hardware handshaking typically utilizes the Request to Send (RTS) and Clear to Send (CTS) pins to control data flow. Software handshaking uses control characters (Xon and Xoff) to control data flow. To learn more about hardware and software handshaking, refer to "Using Control Pins" on page 6-39.

You can return the value of the CTS pin with the `PinStatus` property. You can specify the value of the RTS pin with the `RequestToSend` property. However, if `FlowControl` is `hardware`, and you specify a value for `RequestToSend`, then that value might not be honored.

If you set the `FlowControl` property to `hardware` on a serial object, and a hardware connection is not detected, the `fwrite` and the `fprintf` functions will return an error message. This occurs if a device is not connected, or a connected device is not asserting that is ready to receive data. Check you remote device's status and flow control settings to see if hardware flow control is causing errors in MATLAB.

Notes If you want to check to see if the device is asserting that it is ready to receive data, set the `FlowControl` to `none`. Once you connect to the device check the `PinStatus` structure for `ClearToSend`. If `ClearToSend` is `off`, there is a problem on the remote device side. If `ClearToSend` is `on`, there is a hardware `FlowControl` device prepared to receive data and you can execute `fprintf` and `fwrite`.

Although you might be able to configure your instrument for both hardware handshaking and software handshaking at the same time, the toolbox does not support this behavior.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	String

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

State of GPIB handshake lines

Description

HandshakeStatus is a structure array that contains the fields DataValid, NotDataAccepted, and NotReadyForData. These fields indicate the state of the Data Valid (DAV), Not Data Accepted (NDAC) and Not Ready For Data (NRFD) GPIB lines, respectively.

HandshakeStatus can be on or off for any of these fields. A value of on indicates the associated line is asserted. A value of off indicates the associated line is unasserted.

Characteristics

Usage	GPIB
Read only	Never
Data type	Structure

Values

on	The associated handshake line is asserted
off	The associated handshake line is unasserted

The default value is instrument dependent.

HardwareAssets

Array of hardware assets contained in IVI configuration store

Description

HardwareAssets specifies all hardware assets in the IVI configuration store, each hardware asset referencing an IOResourceDescriptor.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

Values

The default value is empty.

See Also

Properties

DriverSessions, SoftwareModules

HwIndex

Hardware index of device group object

Description

Every device group object contained by a device object has an associated hardware index that is used to reference that device group object. For example, to configure property values for an individual device group object, you must reference the group object through its property name and the appropriate `HwIndex` value.

`HwIndex` provides a convenient way to programmatically access device group objects.

Characteristics

Usage	Device Group
Read only	Always
Data type	Double

Values

The default value is defined at the device group object creation.

See Also

Properties

`HwName`

HwName

Hardware name of device group object

Description

Every device group object contained by a device object has an associated hardware name that can be used to reference that device group object.

HwName provides a convenient way to programmatically access device group objects.

Characteristics

Usage	Device Group
Read only	Always
Data type	String

Values

The default value is defined at the device group object creation.

See Also

Properties

HwIndex

InputBufferSize

Specify size of input buffer in bytes

Description

You configure `InputBufferSize` as the total number of bytes that can be stored in the software input buffer during a read operation.

A read operation is terminated if the amount of data stored in the input buffer equals the `InputBufferSize` value. You can read text data with the `fgetl`, `fgets`, or `fscanf` functions. You can read binary data with the `fread` function.

You can configure `InputBufferSize` only when the instrument object is disconnected from the instrument. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

If you configure `InputBufferSize` while there is data in the input buffer, then that data is flushed.

Characteristics

Usage	Any instrument object
Read only	While open
Data type	Double

Values

The default value is `512`.

See Also

Functions

`fclose`, `fgetl`, `fgets`, `fopen`, `fread`, `fscanf`

Properties

Status

InputDatagramPacketSize

Specify length of data received in a datagram

Description

Specify the length of the data received in a datagram. The size is the number of bytes of the packet's data buffer used to receive data.

If the data in a datagram packet is larger than the `InputDatagramPacketSize` the incoming data is truncated and some data is lost.

Characteristics

Usage	UDP
Read only	When open
Data type	Double

Values

You can specify a size, in bytes, between 1 and 65,535. The default value is 512.

See Also

Functions

`udp`

Properties

`OutputDatagramPacketSize`

InstrumentModel

Instrument model that object connects to

Description

`InstrumentModel` returns the information returned by the instrument identification command, e.g., `*IDN?`, `*ID?`. The instrument identification command is defined by the instrument driver.

Characteristics

Usage	Device
Read only	Always
Data type	String

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.

See Also

Functions

`connect`, `get`

Interface

Interface object that communicates with instrument

Description

If `DriverType` is `MATLAB Instrument Driver`, then `Interface` is the interface object used to communicate with the instrument. If `DriverType` is `VXIplug&play` or `IVI-C`, then `Interface` is the handle to the VISA session that is used to communicate with the instrument. If `DriverType` is `IVI-COM`, `Interface` is the handle to the IVI driver's default interface.

Characteristics

Usage	Device
Read only	Always
Data type	String

Values

`Interface` is defined at device object creation.

See Also

Properties

`DriverType`, `LogicalName`, `RsrcName`

InterfaceIndex

Specify USB interface number

Description

You configure `InterfaceIndex` to be the USB interface number.

The `Name` and `RsrcName` properties are automatically updated to reflect the `InterfaceIndex` value.

You can configure `InterfaceIndex` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-USB
Read only	While open
Data type	Double

See Also

Functions

`fclose`

Properties

`Name`, `RsrcName`

InterruptFcn

Specify callback function to execute when interrupt event occurs

Description

You configure `InterruptFcn` to execute a callback function when an interrupt event occurs. An interrupt event is generated when a VXI bus signal or a VXI bus interrupt is received from the instrument.

Note An interrupt event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and an interrupt event occurs, the record file records

- The event type as `Interrupt`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Characteristics

Usage	VISA-VXI
Read only	Never
Data type	String

Values

The default value is an empty string.

See Also

Functions

record

Properties

RecordStatus

LANName

Specify LAN device name

Description

You configure `LANName` to be the LAN (Local Area Network) device name.

The `Name` and `RsrcName` properties are automatically updated to reflect the `LANName` value.

You can configure `LANName` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-TCPIP
Read only	While open
Data type	String

See Also

Functions

`fclose`

Properties

`Name`, `RsrcName`

LocalHost

Specify local host

Description

`LocalHost` specifies the local host name or the IP dotted decimal address. An example dotted decimal address is 144.212.100.10. If you have only one address or you do not specify this property, the object uses the default IP address when you connect to the hardware with the `fopen` function.

You can configure `LocalHost` only when the object is disconnected from the hardware. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	TCPIP, UDP
Read only	While open
Data type	String

Values

The default value is an empty string.

See Also

Functions

`fclose`, `fopen`, `tcpip`, `udp`

Properties

`LocalPort`, `RemoteHost`, `Status`

LocalPort

Specify local host port for connection

Description

You configure `LocalPort` to be the port value of the local host. The default value is `[ ]`.

If `LocalPortMode` is set to `auto` or if `LocalPort` is `[ ]`, the property is assigned any free port when you connect the object to the hardware with the `fopen` function. If `LocalPortMode` is set to `manual`, the specified `LocalPort` value is used when you issue `fopen`. If you explicitly configure `LocalPort`, `LocalPortMode` is automatically set to `manual`.

You can configure `LocalPort` only when the object is disconnected from the hardware. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	TCPIP, UDP
Read only	While open
Data type	Double

Values

The default value is `[ ]`.

See Also

Functions

`fclose`, `fopen`, `tcpip`, `udp`

Properties

`LocalHost`, `LocalPortMode`, `Status`

LocalPortMode

Specify local host port selection mode

Description

`LocalPortMode` specifies the selection mode for the `LocalPort` property when you connect a TCPIP or UDP object.

If `LocalPortMode` is set to `auto`, the MATLAB workspace uses any free local port. If `LocalPortMode` is set to `manual`, the specified `LocalPort` value is used when you issue the `fopen` function. If you explicitly specify a value for `LocalPort`, `LocalPortMode` is automatically set to `manual`.

Characteristics

Usage	TCPIP, UDP
Read only	While open
Data type	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`

LogicalAddress

Specify logical address of VXI instrument

Description

For VISA-VXI and VISA-GPIB-VXI objects, you configure `LogicalAddress` to be the logical address of the VXI instrument. You must include the logical address as part of the resource name during object creation using the `visa` function.

The `Name` and `RsrcName` properties are automatically updated to reflect the `LogicalAddress` value.

You can configure `LogicalAddress` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	While open
Data type	Double

Values

The value is defined when the instrument object is created.

Examples

This example creates a VISA-VXI object associated with chassis 4 and logical address 1, and then returns the logical address.

```
vv = visa('agilent','VXI4::1::INSTR');
vv.LogicalAddress
```

```
ans =
     1
```

See Also

Functions

`fclose`, `visa`

Properties

`Name`, `RsrcName`, `Status`

LogicalName

Specify description of interface used to communicate with instrument

Description

For device objects with a `DriverType` property value of `MATLAB Instrument Driver`, the `LogicalName` property specifies the type of interface used to communicate with the instrument. For example, a `LogicalName` of `GPIB0-2` indicates that communication is through a GPIB board at index 0 with an instrument at primary address 2.

For device objects with a `DriverType` property value of `VXIplug&play`, the `LogicalName` is the resource name used to communicate with the instrument.

For device objects with a `DriverType` property value of `IVI-C` 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`, `RsrcName`

LogicalNames

Array of logical names contained in IVI configuration store

Description

Each entry in `LogicalNames` identifies a logical name in the IVI configuration store. Each logical name references a driver session in the configuration store, and is used in creating device objects with the `icdevice` function.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

See Also

Functions

`icdevice`

Properties

`DriverSessions`

ManufacturerID

Specify manufacturer ID of USB instrument

Description

You configure `ManufacturerID` to be the manufacturer ID of the USB instrument.

The `Name` and `RsrcName` properties are automatically updated to reflect the `ManufacturerID` value.

You can configure `ManufacturerID` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-USB
Read only	While open
Data type	String

See Also

Functions

`fclose`

Properties

`Name`, `RsrcName`

MappedMemoryBase

Base memory address of mapped memory

Description

`MappedMemoryBase` is the base address of the mapped memory used for low level read and write operations.

The memory address is returned as a 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.

```
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.

```
memmap(vv,'A16',0,16)
vv.MappedMemoryBase
ans =
     16737610H
```

See Also

Functions

`memmap`, `memunmap`

Properties

`MappedMemorySize`

MappedMemorySize

Size of mapped memory for low-level read and write operations

Description

MappedMemorySize indicates the amount of memory mapped for low-level read and write operations.

Use the memmap function to map the specified amount of memory in the specified address space (A16, A24, or A32) with the specified offset. Use the memunmap function to unmap the memory space.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	Double

Values

The default value is 0.

Examples

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

```
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.

```
memmap(vv,'A16',0,16)
```

```
vv.MappedMemorySize
ans =
     16
```

See Also

Functions

`memmap`, `memunmap`

Properties

`MappedMemoryBase`

MasterLocation

Full pathname of master configuration store file

Description

`MasterLocation` identifies the master (default) configuration store location.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	String

Values

The default value is set at IVI installation.

See Also

Properties

`ActualLocation`, `ProcessLocation`

MemoryBase

Base address of A24 or A32 space

Description

`MemoryBase` indicates the base address of the A24 or A32 space. The value is returned as a 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.

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

```
fopen(vv)
```

The `MemorySpace` property indicates that the A24 memory space is supported.

```
vv.MemorySpace
ans =
A16/A24
```

The base address of the A24 space is

```
vv.MemoryBase
ans =
'200000H'
```

See Also

Properties

```
MemorySpace
```

MemoryIncrement

Specify whether VXI register offset increments after data is transferred

Description

You can configure `MemoryIncrement` to be `block` or `FIFO`. If `MemoryIncrement` is `block`, the `memread` and `memwrite` functions increment the offset after every read and write operation, and data is transferred from or to consecutive memory elements. If `MemoryIncrement` is `FIFO`, the `memread` and `memwrite` functions do not increment the VXI register offset, and data is always read from or written to the same memory element.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Never
Data type	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.

```
v = visa('ni','VXI0::8::INSTR');
fopen(v)
```

Configure the hardware for a FIFO read and write operation.

```
v.MemoryIncrement = 'FIFO'
```

Write two values to the VXI register starting at offset 16. Because `MemoryIncrement` is `FIFO`, the VXI register offset does not change and both values are written to offset 16.

```
memwrite(v,[1984 2000],16,'uint32','A16')
```

Read the value at offset 16. The value returned is the second value written with the `memwrite` function.

```
memread(v,16,'uint32')
ans =
2000
```

Read two values starting at offset 16. Note that both values are read at offset 16.

```
memread(v,16,'uint32','A16',2);
ans =
2000
2000
```

Configure the hardware for a block read and write operation.

```
v.MemoryIncrement = 'block'
```

Write two values to the VXI register starting at offset 16. The first value is written to offset 16 and the second value is written to offset 20 because a `uint32` value consists of four bytes.

```
memwrite(v,[1984 2000],16,'uint32','A16')
```

Read the value at offset 16. The value returned is the first value written with the `memwrite` function.

```
memread(v,16,'uint32')
ans =
1984
```

Read two values starting at offset 16. The first value is read at offset 16 and the second value is read at offset 20.

```
memread(v,16,'uint32','A16',2);
ans =
1984
2000
```

See Also

Functions

mempeek, mempoke, `memread`, `memwrite`

MemorySize

Size of memory requested in A24 or A32 address space

Description

MemorySize indicates the size of the memory requested by the instrument in the A24 or A32 address space.

Some instruments use the A24 or A32 address space when the 64 bytes of A16 space are not enough for performing necessary functions. An instrument cannot use both the A24 and A32 address space. The address space is given by the MemorySpace property. If MemorySpace is A16, then MemorySize is 0.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	Double

Values

The default value is 0.

Examples

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

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

The MemorySpace property indicates that the A24 memory space is supported.

```
vv.MemorySpace
```

```
ans =
A16/A24
```

The size of the A24 space is

```
vv.MemorySize
ans =
262144
```

See Also

Properties

```
MemorySpace
```

MemorySpace

Address space used by instrument

Description

`MemorySpace` indicates the address space requested by the instrument. `MemorySpace` can be `A16`, `A16/A24`, or `A16/A32`. If `MemorySpace` is `A16`, the instrument uses only the A16 address space. If `MemorySpace` is `A16/A24`, the instrument uses the A16 and A24 address space. If `MemorySpace` is `A16/A32`, the instrument uses the A16 and A32 address space.

All VXI instruments have an A16 address space that is 16 bits wide. There are also 24- and 32-bit wide address spaces known as A24 and A32, respectively. Some instruments use this memory when the 64 bytes of A16 space are not enough for performing necessary functions. An instrument cannot use both the A24 and A32 address space.

The size of the memory is given by the `MemorySize` property. The base address of the memory is given by the `MemoryBase` property.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	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.

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

Return the memory space supported by the instrument.

```
vv.MemorySpace
ans =
A16/A24
```

This value indicates that the instrument supports A24 memory space in addition to the A16 memory space.

See Also

Properties

MemoryBase, MemorySize

ModelCode

Specify model code of USB instrument

Description

You configure `ModelCode` to be the model code of the USB instrument.

The `Name` and `RsrcName` properties are automatically updated to reflect the `ModelCode` value.

You can configure `ModelCode` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-USB
Read only	While open
Data type	String

See Also

Functions

`fclose`

Properties

`Name`, `RsrcName`

Name

Specify descriptive name for instrument object

Description

You configure `Name` to be a descriptive name for an instrument object.

When you create an instrument object, a descriptive name is automatically generated and stored in `Name`. However, you can change this value at any time. As shown below, the components of `Name` reflect the instrument object type and the input arguments you supply to the creation function.

Instrument Object	Default Value of Name
GPIB	`GPIB` and `BoardIndex-PrimaryAddress-SecondaryAddress`
serial port	`Serial` and `Port`
TCPIP	`TCPIP` and `RemoteHost`
UDP	`UDP` and `RemoteHost`
VISA-serial	`VISA-Serial` and `Port`
VISA-GPIB	`VISA-GPIB` and `BoardIndex-PrimaryAddress-SecondaryAddress`
VISA-VXI	`VISA-VXI` and `ChassisIndex-LogicalAddress`
VISA-GPIB-VXI	`VISA-GPIB-VXI` and `ChassisIndex-LogicalAddress`
VISA-TCPIP	`VISA-TCPIP` and `BoardIndex-RemoteHost-LANName`
VISA-RSIB	`VISA-RSIB` and `RemoteHost`
VISA-USB	`VISA-USB` and `BoardIndex-ManufacturerID- ModelCode-SerialNumber-InterfaceIndex`

If the secondary address is not specified when a GPIB or VISA-GPIB object is created, then `Name` does not include this component.

If you change the value of any property that is a component of `Name` (for example, `Port` or `PrimaryAddress`), then `Name` is automatically updated to reflect those changes.

Characteristics

Usage	Any instrument object
Read-only	Never
Data type	String

Values

Name is automatically defined at object creation time. The value of Name depends on the specific instrument object you create.

Name (iviconfigurationstore)

Name of IVI configuration server

Description

Name identifies the name of the IVI configuration store server. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	String

NetworkRole

Specify server socket connection

Description

The `NetworkRole` property in the `tcpip` interface enables support for Server Sockets. It uses two values, `client` and `server`, to establish a connection as the client or the server.

The server sockets feature supports binary and ASCII transfers and supports a single remote connection.

Characteristics

Usage	TCPIP
Read only	While open
Data type	String

Values

The default value is `client`.

`client`	Establish a TCP/IP connection as a client (default)
`server`	Establish a TCP/IP connection as a server

See Also

Functions

`fclose`, `fopen`, `tcpip`

How To

"Using TCP/IP Server Sockets" on page 7-52

ObjectVisibility

Control access to instrument object

Description

The `ObjectVisibility` property provides a way for application developers to prevent end-user access to the instrument objects created by their application. When an object's `ObjectVisibility` property is set to `off`, `instrfind` and `instrreset` do not return or delete those objects.

Objects that are not visible are still valid. If you have access to the object (for example, from within the file that creates it), then you can set and get its properties and pass it to any function that operates on instrument objects.

Characteristics

Usage	Any instrument object
Read only	Never
Data type	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`:

```
g = gpib('keithley',0,2,'ObjectVisibility','off');
instrfind
ans =
     []
```

However, since the object is in the workspace (g), you can access it.

```
g.ObjectVisibility
ans =

 off
```

See Also

Functions

`instrfind`, `instrfindall`, `instrreset`

OutputBufferSize

Specify size of output buffer in bytes

Description

You configure `OutputBufferSize` as the total number of bytes that can be stored in the software output buffer during a write operation.

An error occurs if the output buffer cannot hold all the data to be written. You write text data with the `fprintf` function. You write binary data with the `fwrite` function.

You can configure `OutputBufferSize` only when the instrument object is disconnected from the instrument. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	Any instrument object
Read only	While open
Data type	Double

Values

The default value is `512`.

See Also

Functions

`fprintf`, `fwrite`

Properties

```
Status
```

OutputDatagramPacketSize

Specify length of data sent in a datagram

Description

Specify the length of the data sent in a datagram. The size is the number of bytes of the packet's data buffer used to send data.

If the data in a datagram packet is larger than the packet size the target device receives, some data is lost.

Characteristics

Usage	UDP
Read only	When open
Data type	Double

Values

You can specify a size, in bytes, between 1 and 65,535. The default value is 512.

See Also

Functions

`udp`

Properties

`InputDatagramPacketSize`

OutputEmptyFcn

Specify callback function to execute when output buffer is empty

Description

You configure `OutputEmptyFcn` to execute a callback function when an output-empty event occurs. An output-empty event is generated when the last byte is sent from the output buffer to the instrument.

Note An output-empty event can be generated only for asynchronous write operations.

If the `RecordStatus` property value is `on`, and an output-empty event occurs, the record file records this information:

- The event type as `OutputEmpty`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

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
```

Parent

Parent (device object) of device group object

Description

The parent of a device group object is defined as the device object that contains the device group object.

You can create a copy of the device object containing a particular device group object by returning the value of `Parent`. This copy can be treated like any other device object. For example, you can configure property values, connect to the instrument, and so on.

Characteristics

Usage	Device group
Read only	Always
Data type	Device object

Values

`Parent` is defined at device object creation.

Parity

Specify type of parity checking

Description

You can configure `Parity` to be `none`, `odd`, `even`, `mark`, or `space`. If `Parity` is `none`, parity checking is not performed and the parity bit is not transmitted. If `Parity` is `odd`, the number of mark bits (1s) in the data is counted, and the parity bit is asserted or unasserted to obtain an odd number of mark bits. If `Parity` is `even`, the number of mark bits in the data is counted, and the parity bit is asserted or unasserted to obtain an even number of mark bits. If `Parity` is `mark`, the parity bit is asserted. If `Parity` is `space`, the parity bit is unasserted.

Parity checking can detect errors of one bit only. An error in two bits might cause the data to have a seemingly valid parity, when in fact it is incorrect. To learn more about parity checking, refer to "The Parity Bit" on page 6-10.

In addition to the parity bit, the serial data format consists of a start bit, between five and eight data bits, and one or two stop bits. You specify the number of data bits with the `DataBits` property, and the number of stop bits with the `StopBits` property.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	String

Values

{none}	No parity checking
odd	Odd parity checking
even	Even parity checking
mark	Mark parity checking

space   Space parity checking

See Also

Properties

`DataBits`, `StopBits`

PinStatus

State of CD, CTS, DSR, and RI pins

Description

`PinStatus` is a structure array that contains the fields `CarrierDetect`, `ClearToSend`, `DataSetReady` and `RingIndicator`. These fields indicate the state of the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) and Ring Indicator (RI) pins, respectively. Refer to "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 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 "Connecting Two Modems" on page 6-39 for an example that uses `PinStatus`.

Characteristics

Usage	Serial port, VISA-serial
Read only	Always
Data type	Structure

Values

`off`	The associated pin is asserted
`on`	The associated pin is asserted

The default value is instrument dependent.

See Also

Properties

DataTerminalReady, PinStatusFcn, RequestToSend

PinStatusFcn

Specify callback function to execute when CD, CTS, DSR, or RI pin changes state

Description

You configure `PinStatusFcn` to execute a callback function when a pin status event occurs. A pin status event occurs when the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) or Ring Indicator (RI) pin changes state. A serial port pin changes state when it is asserted or unasserted. Information about the state of these pins is recorded in the `PinStatus` property.

Note A pin status event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a pin status event occurs, the record file records this information:

* The event type as `PinStatus`
* The pin that changed its state, and pin state as either `on` or `off`
* The time the event occurred using the format day-month-year hour:minute:second:millisecond

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`

Port

Specify platform-specific serial port name

Description

You configure `Port` to be the name of a serial port on your platform. `Port` specifies the physical port associated with the object and the instrument.

When you create a serial port or VISA-serial object, `Port` is automatically assigned the port name specified for the `serial` or `visa` function.

You can configure `Port` only when the object is disconnected from the instrument. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	Serial port, VISA-serial
Read only	While open
Data type	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.

```
s = serial('COM1')
vs = visa('ni','ASRL1::INSTR')
```

The `Port` property values are given below.

```
[s vs].Port
ans =
    'COM1'
    'ASRL1'
```

See Also

Functions

`fclose`, `serial`, `visa`

Properties

`Name`, `RsrcName`, `Status`

PrimaryAddress

Specify primary address of GPIB instrument

Description

For GPIB and VISA-GPIB objects, you configure `PrimaryAddress` to be the GPIB primary address associated with your instrument. The primary address can range from 0 to 30, and you must specify it during object creation using the `gpib` or `visa` function. For VISA-GPIB-VXI objects, `PrimaryAddress` is read-only, and the value is returned automatically by the VISA interface after the object is connected to the instrument with the `fopen` function.

For GPIB and VISA-GPIB objects, the `Name` property is automatically updated to reflect the `PrimaryAddress` value. For VISA-GPIB objects, the `RsrcName` property is automatically updated to reflect the `PrimaryAddress` value.

You can configure `PrimaryAddress` only when the GPIB or VISA-GPIB object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-GPIB-VXI
Read only	While open (GPIB, VISA-GPIB), always (VISA-GPIB-VXI)
Data type	Double

Values

`PrimaryAddress` can range from 0 to 30. The value is determined when the instrument object is created.

Examples

This example creates a VISA-GPIB object associated with board 0, primary address 1, and secondary address 8, and then returns the primary address.

```
vg = visa('agilent','GPIB0::1::8::INSTR');
vg.PrimaryAddress
ans =
     1
```

See Also

Functions

`fclose`, `gpib`, `visa`

Properties

`Name`, `RsrcName`, `Status`

ProcessLocation

Configuration store file for process to use if master configuration store is not used

Description

`ProcessLocation` identifies an IVI configuration store being used as an alternative to the master configuration store. The use of an alternative is particular to each `iviconfigurationstore` object, and is specified when the object is created.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	String

Values

The default value is an empty string.

See Also

Functions

`iviconfigurationstore`

Properties

`ActualLocation, MasterLocation`

PublishedAPIs

Array of published APIs in IVI configuration store

Description

PublishedAPIs identifies the published APIs in the IVI configuration store server. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

ReadAsyncMode

Specify whether asynchronous read operation is continuous or manual

Description

You can configure `ReadAsyncMode` to be `continuous` or `manual`. If `ReadAsyncMode` is `continuous`, the object continuously queries the instrument to determine if data is available to be read. If data is available, it is automatically read and stored in the input buffer. If issued, the `readasync` function is ignored.

If `ReadAsyncMode` is `manual`, the object will not query the instrument to determine if data is available to be read. Instead, you must manually issue the `readasync` function to perform an asynchronous read operation. Because `readasync` checks for the terminator, this function can be slow. To increase speed, you should configure `ReadAsyncMode` to `continuous`.

Note If the instrument is ready to transmit data, then it will do so regardless of the `ReadAsyncMode` value. Therefore, if `ReadAsyncMode` is `manual` and a read operation is not in progress, then data can be lost. To guarantee that all transmitted data is stored in the input buffer, you should configure `ReadAsyncMode` to `continuous`.

You can determine the amount of data available in the input buffer with the `BytesAvailable` property. For either `ReadAsyncMode` value, you can bring data into the MATLAB workspace with one of the synchronous read functions such as `fscanf`, `fgetl`, `fgets`, or `fread`.

Characteristics

Usage	Serial port, TCPIP, UDP, VISA-serial
Read only	Never
Data type	String

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`

RecordDetail

Specify amount of information saved to record file

Description

You can configure `RecordDetail` to be `compact` or `verbose`. If `RecordDetail` is `compact`, the number of values written to the instrument, the number of values read from the instrument, the data type of the values, and event information are saved to the record file. If `RecordDetail` is `verbose`, the data transferred to and from the instrument is also saved to the record file.

The verbose record file structure is shown in "Recording Information to Disk" on page 16-10.

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

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 16-7 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** on a Windows machine associated with the serial port COM1.

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

Specify the record filename with the `RecordName` property, configure `RecordMode` to `index`, and initiate recording.

```
s.RecordName = 'myrecord.txt';
s.RecordMode = 'index';
record(s)
```

The record filename is automatically updated with an indexed filename after recording is turned off.

```
record(s,'off')
s.RecordName
ans =
myrecord01.txt
```

Disconnect **s** from the instrument, and remove **s** from memory and from the MATLAB workspace.

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

See Also

Functions

`record`

Properties

`RecordDetail`, `RecordName`, `RecordStatus`

RecordName

Specify name of record file

Description

You configure `RecordName` to be the name of the record file. You can specify any value for `RecordName` — including a directory path — provided the filename is supported by your operating system.

The MATLAB software supports any filename supported by your operating system. However, if you access the file through the MATLAB workspace, you might need to specify the filename using single quotes. For example, suppose you name the record file `my record.txt`. To type this file at the MATLAB Command Window, you must include the name in quotes.

```
type('my record.txt')
```

You can specify whether data and event information are saved to one disk file or to multiple disk files with the `RecordMode` property. If `RecordMode` is `index`, then the filename follows a prescribed set of rules. Refer to "Specifying a File Name" on page 16-7 for a description of these rules.

You can configure `RecordName` only when the object is not recording. You terminate recording with the `record` function. An object that is not recording has a `RecordStatus` property value of `off`.

Characteristics

Usage	Any instrument object
Read only	While recording
Data type	String

Values

The default record file name is `record.txt`.

See Also

Functions

`record`

Properties

`RecordDetail`, `RecordMode`, `RecordStatus`

RecordStatus

Status of whether data and event information are saved to record file

Description

You can configure `RecordStatus` to be `off` or `on` with the `record` function. If `RecordStatus` is `off`, then data and event information are not saved to a record file. If `RecordStatus` is `on`, then data and event information are saved to the record file specified by `RecordName`.

Use the `record` function to initiate or complete recording. `RecordStatus` is automatically configured to reflect the recording state.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	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

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

Specify remote host port for connection

Description

You can configure `RemotePort` to be any port number between `1` and `65535`. The default value is `80` for TCPIP objects and `9090` for UDP objects.

For TCPIP objects, you can configure `RemotePort` only when the object is disconnected from the hardware. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

For UDP objects, you can configure `RemotePort` at any time.

Characteristics

Usage	TCPIP, UDP
Read only	While open (TCPIP), never (UDP)
Data type	Double

Values

Any port number between `1` and `65535`. The default value is `80` for TCPIP objects and `9090` for UDP objects.

See Also

Functions

`fclose`, `fopen`, `tcpip`, `udp`

Properties

`RemoteHost`, `LocalPort`, `Status`

RequestToSend

Specify state of RTS pin

Description

You can configure `RequestToSend` to be `on` or `off`. If `RequestToSend` is `on`, the Request to Send (RTS) pin is asserted. If `RequestToSend` is `off`, the RTS pin is unasserted.

In normal usage, the RTS and Clear to Send (CTS) pins work together, and are used as standard handshaking pins for data transfer. In this case, RTS and CTS are automatically managed by the DTE and DCE. However, there is nothing in the RS-232, or the RS-484 standard that states the RTS pin must to be used in any specific way. Therefore, if you manually configure the `RequestToSend` value, it is probably for nonstandard operations.

If your instrument does not use hardware handshaking in the standard way, and you need to manually configure `RequestToSend`, then you should configure the `FlowControl` property to `none`. Otherwise, the `RequestToSend` value that you specify might not be honored. Refer to your instrument documentation to determine its specific pin behavior.

You can return the value of the CTS pin with the `PinStatus` property. Handshaking is described in "The Control Pins" on page 6-7.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	String

Values

{on}	The RTS pin is asserted.

off   The RTS pin is unasserted.

See Also

Properties

`FlowControl`, `PinStatus`

Revision

IVI configuration store version

Description

Revision identifies the version of the IVI configuration store. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	String

RsrcName

Resource name for VISA instrument

Description

RsrcName indicates the resource name for a VISA instrument. When you create a VISA object, RsrcName is automatically assigned the value specified in the visa function.

The resource name is a symbolic name for the instrument. The resource name you supply to visa depends on the interface and has the format shown below. The components in brackets are optional and have a default value of 0, except port_number, which has a default value of 1.

Interface	Resource Name
VXI	VXI[chassis]::VXI_logical_address::INSTR
GPIB-VXI	GPIB-VXI[chassis]::VXI_logical_address::INSTR
GPIB	GPIB[board]::primary_address[::secondary_address]::INSTR
TCPIP	TCPIP[board]::remote_host[::lan_device_name]::INSTR
RSIB	RSIB::remote_host::INSTR
Serial	ASRL[port_number]::INSTR
USB	USB[board]::manid::model_code::serial_No[::interface_No]::INSTR

If you change the BoardIndex, ChassisIndex, InterfaceIndex, LANName, LogicalAddress, ManufacturerID, ModelCode, Port, PrimaryAddress, RemoteHost, SecondaryAddress, or SerialNumber property value, RsrcName is automatically updated to reflect the change.

Characteristics

Usage	VISA-GPIB, VISA-VXI, VISA-GPIB-VXI, VISA-serial
Read only	Always

Data type   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 = visa('ni','GPIB0::1::INSTR');
```

To create a VISA-VXI object associated with a VXI chassis with index 0 and an instrument with logical address 130, you supply the following resource name to the `visa` function.

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

See Also

Functions

`visa`

Properties

BoardIndex, ChassisIndex, InterfaceIndex, LANName, LogicalAddress,
ManufacturerID, ModelCode, Port, PrimaryAddress, RemoteHost,
SecondaryAddress, SerialNumber

SecondaryAddress

Specify secondary address of GPIB instrument

Description

For GPIB and VISA-GPIB objects, you configure `SecondaryAddress` to be the GPIB secondary address associated with your instrument. You can initially specify the secondary address during object creation using the `gpib` or `visa` function. For VISA-GPIB-VXI objects, `SecondaryAddress` is read-only, and the value is returned automatically by the VISA interface after the object is connected to the instrument with the `fopen` function.

For GPIB objects, `SecondaryAddress` can range from 96 to 126, or it can be 0 indicating that no secondary address is used. For VISA-GPIB objects, `SecondaryAddress` can range from 0 to 30. If your instrument does not have a secondary address, then `SecondaryAddress` is 0.

For GPIB and VISA-GPIB objects, the `Name` property is automatically updated to reflect the `SecondaryAddress` value. For VISA-GPIB objects, the `RsrcName` property is automatically updated to reflect the `SecondaryAddress` value.

You can configure `SecondaryAddress` only when the GPIB or VISA-GPIB object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	GPIB, VISA-GPIB, VISA-GPIB-VXI
Read only	While open (GPIB, VISA-GPIB), always (VISA-GPIB-VXI)
Data type	Double

Values

For GPIB objects, `SecondaryAddress` can range from 96 to 126, or it can be 0. For VISA-GPIB objects, `SecondaryAddress` can range from 0 to 30. The default value is `0`.

Examples

This example creates a VISA-GPIB object associated with board 0, primary address 1, and secondary address 8, and then returns the secondary address.

```
vg = visa('agilent','GPIB0::1::8::INSTR');
vg.SecondaryAddress
ans =
     8
```

See Also

Functions

`fclose`, `gpib`, `visa`

Properties

`Name`, `RsrcName`, `Status`

SerialNumber

Specify index of USB instrument on USB hub

Description

You configure `SerialNumber` to be the index of the USB instrument on the USB hub.

The `Name` and `RsrcName` properties are automatically updated to reflect the `SerialNumber` value.

You can configure `SerialNumber` only when the object is disconnected from the instrument. You disconnect a connected object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Usage	VISA-USB
Read only	While open
Data type	String

See Also

Functions

`fclose`

Properties

`Name`, `RsrcName`

ServerDescription

IVI configuration store server description

Description

ServerDescription contains the description of the IVI configuration store server. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	String

Sessions

Array of driver sessions in IVI configuration store

Description

Sessions identifies the sessions in the IVI configuration store, including the driver sessions.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

See Also

Properties

DriverSessions

Slot

Slot location of VXI instrument

Description

Slot indicates the physical slot of the VXI instrument. Slot can be any value between 0 and 12.

Characteristics

Usage	VISA-VXI, VISA-GPIB-VXI
Read only	Always
Data type	Double

Values

The property value is defined when the instrument object is connected.

SoftwareModules

Array of software modules in IVI configuration store

Description

`SoftwareModules` identifies the software modules in the IVI configuration store. These are installed by the user, but are not configurable. They include instrument drivers.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	Array of Structs

Values

The default value is empty when no software modules are installed.

See Also

Properties

`DriverSessions`, `HardwareAssets`

SpecificationVersion

IVI configuration server specification version that this server revision complies with

Description

`SpecificationVersion` identifies the specification version of the IVI configuration store server. This is not user-configurable.

Characteristics

Usage	IVI configuration store object
Read only	Always
Data type	String

Status

Status of whether object is connected to instrument

Description

`Status` can be `open` or `closed`. If `Status` is `closed`, the object is not connected to the instrument. If `Status` is `open`, the object is connected to the instrument.

Before you can write or read data, you must connect the object to the instrument with the `fopen` function. You use the `fclose` function to disconnect an object from the instrument.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	String

Values

{closed}	The object is not connected to the instrument.
open	The object is connected to the instrument.

See Also

Functions

`fclose`, `fopen`

StopBits

Specify number of bits used to indicate end of byte

Description

You can configure StopBits to be 1, 1.5, or 2 for serial port objects, or 1 or 2 for VISA-serial objects If StopBits is 1, one stop bit is used to indicate the end of data transmission. If StopBits is 2, two stop bits are used to indicate the end of data transmission. If StopBits is 1.5, the stop bit is transferred for 150% of the normal time used to transfer one bit.

Note Both the computer and the instrument must be configured to transmit the same number of stop bits.

In addition to the stop bits, the serial data format consists of a start bit, between five and eight data bits, and possibly a parity bit. You specify the number of data bits with the DataBits property, and the type of parity checking with the Parity property.

Characteristics

Usage	Serial port, VISA-serial
Read only	Never
Data type	double

Values

Serial Port

{1}	One stop bit is transmitted to indicate the end of a byte.
1.5	The stop bit is transferred for 150% of the normal time used to transfer one bit.

2	Two stop bits are transmitted to indicate the end of a byte.

VISA-Serial

{1}	One stop bit is transmitted to indicate the end of a byte.
2	Two stop bits are transmitted to indicate the end of a byte

See Also

Properties

DataBits, Parity

Tag

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

Usage	Any instrument object
Read only	Never
Data type	String

Values

The default value is an empty string.

Examples

Suppose you create a serial port object on a Windows machine associated with the serial port COM1.

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

You can assign `s` a unique label using `Tag`.

```
s.Tag = 'MySerialObj'
```

You can access s in the MATLAB workspace or in a file using the instrfind function and the Tag property value.

```
s1 = instrfind('Tag','MySerialObj');
```

See Also

Functions

instrfind

Terminator

Specify terminator character

Description

For serial, TCPIP, UDP, and VISA-serial objects, you can configure `Terminator` to an integer value ranging from `0` to `127`, to the equivalent ASCII character, or to empty (''). For example, to configure `Terminator` to a carriage return, you specify the value to be `CR` or `13`. To configure `Terminator` to a line feed, you specify the value to be `LF` or `10`. For serial port objects, you can also set `Terminator` to `CR/LF` or `LF/CR`. If `Terminator` is `CR/LF`, the terminator is a carriage return followed by a line feed. If Terminator is `LF/CR`, the terminator is a line feed followed by a carriage return. Note that there are no integer equivalents for these two values.

Additionally, you can set `Terminator` to a 1-by-2 cell array. The first element of the cell is the read terminator and the second element of the cell array is the write terminator.

When performing a write operation using the `fprintf` function, all occurrences of `\n` are replaced with the `Terminator` value. Note that `%s\n` is the default format for `fprintf`. A read operation with `fgetl`, `fgets`, or `fscanf` completes when the `Terminator` value is read. The terminator is ignored for binary operations.

You can also use the terminator to generate a bytes-available event when the `BytesAvailableFcnMode` is set to `terminator`.

Characteristics

Usage	Serial, TCPIP, UDP, VISA-serial
Read only	Never
Data type	ASCII value

Values

An integer value ranging from `0` to `127`, the equivalent ASCII character, or empty (''). For serial port objects, `CR/LF` and `LF/CR` are also accepted values. You specify different read and write terminators as a 1-by-2 cell array.

See Also

Functions

`fgetl`, `fgets`, `fprintf`, `fscanf`

Properties

`BytesAvailableFcnMode`

Timeout

Specify waiting time to complete read or write operation

Description

You configure `Timeout` to be the maximum time (in seconds) to wait to complete a read or write operation.

If a timeout occurs, then the read or write operation aborts. Additionally, if a timeout occurs during an asynchronous read or write operation, then

- An error event is generated.
- The callback function specified for `ErrorFcn` is executed.

Note: Timeouts are rounded upwards to full seconds.

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Double

Values

The default value is 10 seconds.

Note that timeouts are rounded upwards to full seconds.

See Also

Properties

`ErrorFcn`

TimerFcn

Specify callback function to execute when predefined period passes

Description

You configure `TimerFcn` to execute a callback function when a timer event occurs. A timer event occurs when the time specified by the `TimerPeriod` property passes. Time is measured relative to when the object is connected to the instrument with `fopen`.

Note A timer event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a timer event occurs, the record file records this information:

- The event type as `Timer`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

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

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`

TimerPeriod

Specify period between timer events

Description

TimerPeriod specifies the time, in seconds, that must pass before the callback function specified for TimerFcn is called. Time is measured relative to when the object is connected to the instrument with fopen.

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

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Callback function

Values

The default value is 1 second. The minimum value is 0.01 second.

See Also

Functions

fopen

Properties

TimerFcn

TransferDelay

Specify use of TCP segment transfer algorithm

Description

You can configure `TransferDelay` to `on` or `off`. If `TransferDelay` is `on`, small segments of outstanding data are collected and sent in a single packet when acknowledgment (ACK) arrives from the server. If `TransferDelay` is `off`, data is sent immediately to the network.

If a network is slow, you can improve its performance by configuring `TransferDelay` to `on`. However, on a fast network acknowledgments arrive quickly and there is negligible difference between configuring `TransferDelay` to `on` or `off`.

Note that the segment transfer algorithm used by `TransferDelay` is Nagle's algorithm.

Characteristics

Usage	TCPIP
Read only	Never
Data type	String

Values

{on}	Use the TCP segment transfer algorithm.
off	Do not use the TCP segment transfer algorithm.

See Also

Functions

`tcpip`

TransferStatus

Status of whether asynchronous read or write operation is in progress

Description

`TransferStatus` can be `idle`, `read`, `write`, or `read&write`. If `TransferStatus` is `idle`, then no asynchronous read or write operations are in progress. If `TransferStatus` is `read`, then an asynchronous read operation is in progress. If `TransferStatus` is `write`, then an asynchronous write operation is in progress. If `TransferStatus` is `read&write`, then both an asynchronous read and an asynchronous write operation are in progress.

You can write data asynchronously using the `fprintf` or `fwrite` functions. You can read data asynchronously using the `readasync` function, or by configuring `ReadAsyncMode` to `continuous` (serial, TCPIP, UDP, and VISA-serial objects only). For detailed information about asynchronous read and write operations, refer to "Communicating with Your Instrument" on page 2-7.

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.

See Also

Functions

`fprintf`, `fwrite`, `readasync`

Properties

`ReadAsyncMode`

TriggerFcn

Specify callback function to execute when trigger event occurs

Description

You configure `TriggerFcn` to execute a callback function when a trigger event occurs. A trigger event is generated when a trigger occurs in software, or on one of the VXI hardware trigger lines. You configure the trigger type with the `TriggerType` property.

Note A trigger event can be generated at any time during the instrument control session.

If the `RecordStatus` property value is `on`, and a trigger event occurs, the record file records

- The event type as `Trigger`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Characteristics

Usage	VISA-VXI
Read only	Never
Data type	String

Values

The default value is an empty string.

See Also

Functions

`record`

Properties

`RecordStatus`, `TriggerLine`, `TriggerType`

TriggerLine

Specify trigger line on VXI instrument

Description

You can configure `TriggerLine` to be `TTL0` through `TTL7`, `ECL0`, or `ECL1`. You can use only one trigger line at a time.

You can specify the trigger type with the `TriggerType` property. When `TriggerType` is `hardware`, the line triggered is given by the `TriggerLine` value. When the `TriggerType` is `software`, the `TriggerLine` value is ignored.

You execute a trigger for a VISA-VXI object with the `trigger` function.

Characteristics

Usage	VISA-VXI
Read only	Never
Data type	String

Values

`TriggerLine` can be `TTL0` through `TTL7`, `ECL0`, or `ECL1`. The default value is `TTL0`.

See Also

Functions

`trigger`

Properties

`TriggerType`

TriggerType

Specify trigger type

Description

You can configure TriggerType to be software or hardware. If TriggerType is software, then a software trigger is used. If TriggerType is hardware, then the trigger line specified by the TriggerLine property is used.

You execute a trigger for a VISA-VXI object with the trigger function.

Characteristics

Usage	VISA-VXI
Read only	Never
Data type	String

Values

{hardware}	A hardware trigger is used.
software	A software trigger is used.

See Also

Functions

trigger

Properties

TriggerLine

Type

Instrument object type

Description

Type indicates the type of the object. Type is automatically defined after the instrument object is created with the serial, gpib, or visa function.

Using the instrfind function and the Type value, you can quickly identify instrument objects of a given type.

Characteristics

Usage	Any instrument object
Read only	Always
Data type	String

Values

gpib	The object type is GPIB.
serial	The object type is serial port.
tcpip	The object type is TCPIP.
udp	The object type is UDP.
visa-gpib	The object type is VISA-GPIB.
visa-vxi	The object type is VISA-VXI.
visa-gpib-vxi	The object type is VISA-GPIB-VXI.
visa-serial	The object type is VISA-serial.

The value is automatically determined when the instrument object is created.

Examples

Create a serial port object on a Windows machine associated with the serial port COM1. The value of the `Type` property is `serial`, which is the object class.

```
s = serial('COM1');
s.Type
ans =
serial
```

See Also

Functions

instrfind, gpib, serial, tcpip, udp, visa

UserData

Specify data to associate with instrument object

Description

You configure `UserData` to store data that you want to associate with an instrument object. The object does not use this data directly, but you can access it the dot notation.

Characteristics

Usage	Any instrument object
Read only	Never
Data type	Any type

Values

The default value is an empty vector.

Examples

Suppose you create the serial port object on a Windows machine associated with the serial port COM1.

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

You can associate data with `s` by storing it in `UserData`.

```
coeff.a = 1.0;
coeff.b = -1.25;
s.UserData = coeff
```

ValuesReceived

Total number of values read from instrument

Description

`ValuesReceived` indicates the total number of values read from the instrument. The value is updated after each successful read operation, and is set to `0` after the `fopen` function is issued. If the terminator is read from the instrument, then this value is reflected by `ValuesReceived`.

If you are reading data asynchronously, use the `BytesAvailable` property to return the number of bytes currently available in the input buffer.

When performing a read operation, the received data is represented by values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes. Refer to "Output Buffer and Data Flow" on page 3-13 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 on a Windows machine associated with the serial port COM1.

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

If you write the RS232? command, and then read back the response using fscanf, ValuesReceived is 17 because the instrument is configured to send the LF terminator.

```
fprintf(s,'RS232?')
out = fscanf(s)
out =
9600;0;0;NONE;LF
s.ValuesReceived
ans =
    17
```

See Also

Functions

fopen

Properties

BytesAvailable

ValuesSent

Total number of values written to instrument

Description

`ValuesSent` indicates the total number of values written to the instrument. The value is updated after each successful write operation, and is set to `0` after the `fopen` function is issued. If you are writing the terminator, then `ValuesSent` reflects this value.

If you are writing data asynchronously, use the `BytesToOutput` property to return the number of bytes currently in the output buffer.

When performing a write operation, the transmitted data is represented by values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes. Refer to "Output Buffer and Data Flow" on page 3-13 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 on a Windows machine associated with the serial port COM1.

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

```
fopen(s)
```

If you write the `*IDN?` command using the `fprintf` function, then `ValuesSent` is `6` because the default data format is `%s\n`, and the terminator was written.

```
fprintf(s,'*IDN?')
s.ValuesSent
ans =
    6
```

See Also

Functions

```
fopen
```

Properties

```
BytesToOutput
```

Vendor

IVI configuration server vendor

Description

Vendor identifies the vendor of the IVI configuration server. This is not user-configurable.

Characteristics

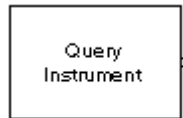|	
Usage	IVI configuration store object
Read only	Always
Data type	String

Block Reference

Query Instrument

Query or read instrument data

Library

Instrument Control Toolbox

Description

The Query Instrument block configures and opens an interface to an instrument, initializes the instrument, and queries the instrument for data. The configuration and initialization happen at the start of the model execution. The block queries the instrument for data during model run time.

The block has no input ports. The block has one output port corresponding to the data received from the instrument.

Dialog Box

Block sample time

> The Block sample time parameter is the only setting outside of the dialog box tabs. The default value is 1. Setting the value to -1 sets the block to inherit timing. A positive value is used as the sample period.
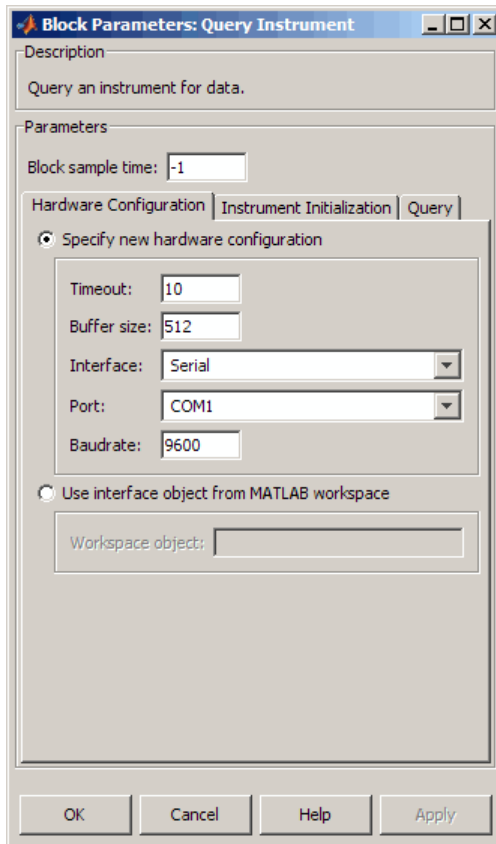
Hardware Configuration Tab

The **Hardware Configuration** tab is where you define the settings for communications with your instrument. You have two choices about establishing an interface:

· Specify a new hardware configuration.

- Use an interface object from the MATLAB workspace.

The following figure shows the **Hardware Configuration** tab set to specify a new hardware configuration using a serial port interface.

Because some parameters apply to multiple interface types, they appear here in alphabetical order.

Baudrate

The rate at which bits are transmitted for the serial or VISA serial interface.

Board index

The index of the board used for GPIB, VISA GPIB, VISA TCPIP, or VISA USB interface to the instrument. See `BoardIndex` property for more information.

Board vendor

The vendor of the GPIB board used for the interface to the instrument. Your choices are Advantech, Agilent, Capital Equipment, Contec, ICS Electronics, 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 the Instrument Control Toolbox software. The previous figure shows a configuration for a serial port interface.

Logical address

The logical address of the VXI instrument. Used for VISA VXI and VISA VXI-GPIB interface types.

Manufacturer ID

The manufacturer ID of the VISA USB instrument. See `ManufacturerID` property for more information.

Model code

The model code of the VISA USB instrument. See `ModelCode` property for more information.

Port

The port for the serial interface: COM1, COM2, etc.

Primary address

The primary address of the instrument on the GPIB.

Remote host

The host name or IP address of the instrument. Used for UDP, TCPIP, or VISA TCPIP interface types.

Remote port

The port on the instrument or remote host used for communication. Used for UDP, TCPIP, or VISA TCPIP interface types.

Secondary address

The secondary address of the instrument on the GPIB.

Serial number

The serial number of the VISA USB instrument defined as a string. See `SerialNumber` property for more information.

Timeout

Time in seconds allowed to complete the query operation.

VISA vendor

The vendor of the VISA used for any of the VISA interface types. Your choices are Agilent, National Instruments, and Tektronix.

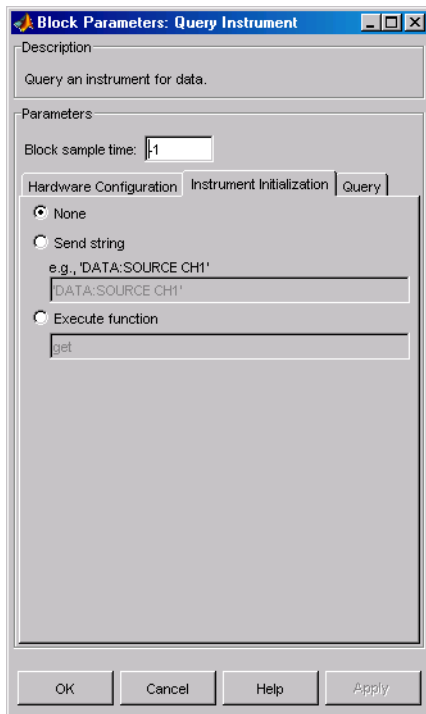**Use interface object from MATLAB workspace**

Select this option to use an interface object from the MATLAB workspace.

Workspace object

Enter the object name that you want to use from the MATLAB workspace.

Instrument Initialization Tab

The **Instrument Initialization** tab is where you define what happens when you first open your connection to the instrument.

None

The default initialization option is none.

Send string

A string sent to the instrument as an instrument command to initialize the instrument or set it up in a known state.

Execute function

Any function that has as its only argument the interface object representing the instrument. You can write this function to include several instrument commands and initialization data.

Query Tab

The **Query** tab is where you define the optional query command, set the format for the response, and define what the block does after the initial instrument response.
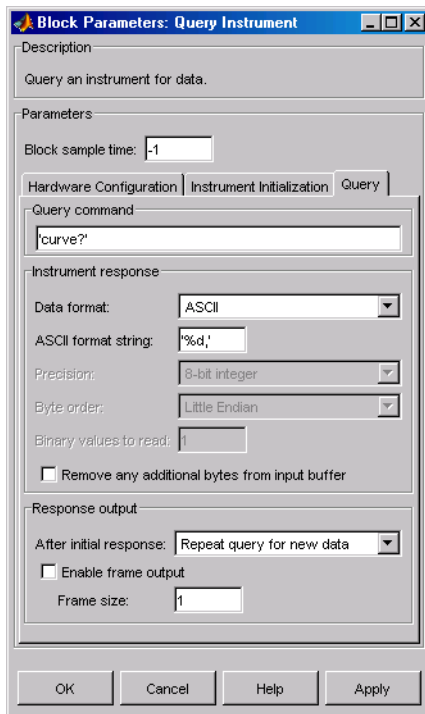
Query command

> This is the query command that is sent to the instrument. It is usually a request for instrument status or data. *This command is optional*—if you are retrieving information or data from the instrument and no query command is necessary to do that, you can leave this field blank.

Data format

> Your options are ASCII, Binary, or Binblock (binary block — the binblock format is described in the binblockwrite function reference page).

ASCII format string

> Available only when the format is ASCII, this defines the format string for the data. For a list of formats, see the fscanf function.

Precision

> Used for binary or binblock format. Your options are:

- 8-bit integer (default)

- `16-bit integer`
- `32-bit integer`
- `8-bit unsigned integer`
- `16-bit unsigned integer`
- `32-bit unsigned integer`
- `32-bit float`
- `64-bit float`

Byte order

When using binary or binblock format with more than 8 bits, you can specify the instrument's byte order for the data. Your options are `Big Endian` or `Little Endian`.

Binary values to read

Used for binary format. Specify the number of binary values to read from the instrument.

Remove additional bytes from input buffer

Select this option if you want to remove any additional bytes from the input buffer before querying.

After initial response

This defines the action to take after the first response from the instrument. Your options are Repeat query for new data, Recycle original data, Hold final value, Output zero, or Stop simulation.

Enable frame output

A frame is a sequence of samples combined into a single vector. In frame-based processing all the samples in a frame are processed simultaneously. In sample-based processing, samples are processed one at a time. The advantage of frame-based processing is that it can greatly increase the speed of a simulation. For example, you might use frames if you are reading a waveform from your instrument rather than a single-point measurement.

Frame size

Frame size determines the number of samples in a frame.

Note Hardware information shown in the dialog box is determined and cached when you first open the dialog box. To refresh the display with new values, restart MATLAB.

See Also

`Serial Configuration`, `Serial Receive`, `Serial Send`, `TCP/IP Receive`, `TCP/IP Send`, `To Instrument`, `UDP Receive`, `UDP Send`
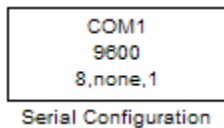
Serial Configuration

Configure parameters for serial port

Library

Instrument Control Toolbox

Description

```
COM1
9600
8,none,1
```
Serial Configuration

The Serial Configuration block configures parameters for a serial port that you can use to send and receive data. You must set the parameters of your serial port before you set up the Serial Receive and the Serial Send block.

You must first specify the configuration of your serial port before you configure the Serial Receive and Serial Send blocks. The Receive and Send blocks will prompt you to add a Configuration block to configure your serial port properties.

Note: You need a license for both the Instrument Control Toolbox and Simulink software to use this block.

Dialog Box

Use the Block Parameters dialog box to select your serial port configuration options.
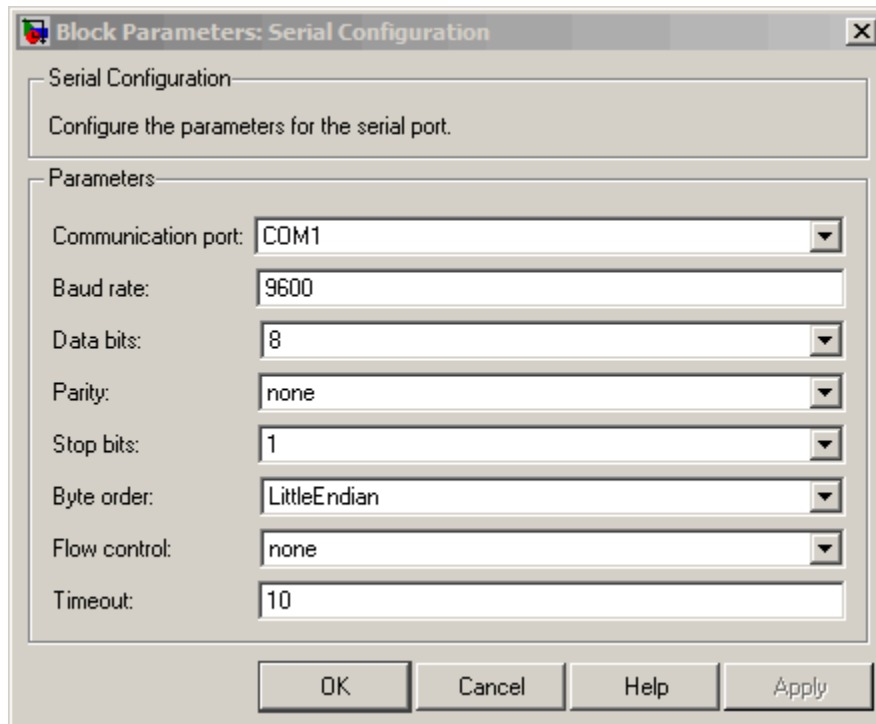
Communication port

   Specify the serial port to configure. You have to select an available port from the list. By default no port is selected and this field displays **<Please select a port...>**. Use this configured port in your Serial Send and Serial Receive blocks. Each Serial Send and Receive block must have a configured serial configuration. If you use multiple serial ports in your simulation, you must configure each port using a separate serial configuration block.

Baud rate

   Specify the rate at which bits are transmitted for the serial interface. Default value is 9600.

Data bits

   Specify the number of data bits to transmit over the serial interface. Default value is **8** and other available values are **5**, **6**, and **7**.

Parity

Specify how you want to check parity bits in the data bits transmitted via the serial port. By default this is set to **none**, and the available values are:

- **none** — Where no parity checking is done.
- **even** — Where parity bit is set to 0 if the number of ones in a given set of bits is even.
- **odd** — Where parity bit is set to 1 if the number of ones in a given set of bits is odd.
- **mark** — Where parity bit is always set to 1.
- **space** — Where parity bit is always set to 0.

Stop bits

Specify the number of bits used to indicate the end of a byte. The number of data bits you select determines the choices available for stop bits. If you select data bits **6**, **7**, or **8**, then the default value is **1** and the other available choice is **2**. If you select data bit **5**, then the only choice available is **1.5**.

Byte order

Specify the byte order as `littleEndian` (default) or `bigEndian`. If byte order is `littleEndian`, then the instrument stores the first byte in the first memory address. If byte order is `bigEndian`, then the instrument stores the last byte in the first memory address.

You should configure byte order to the appropriate value for your instrument before performing a read or write operation. Refer to your instrument documentation for information about the order in which it stores bytes.

Flow control

Specify the process of managing the rate of data transmission on your serial port. Choose **none** to have no flow control or **hardware** to let your hardware determine the flow control.

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).

See Also

Query Instrument, Serial Receive, Serial Send, TCP/IP Receive, TCP/IP Send, To Instrument, UDP Receive, UDP Send
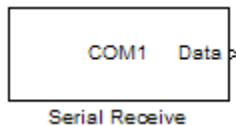
Serial Receive

Receive binary data over serial port

Library
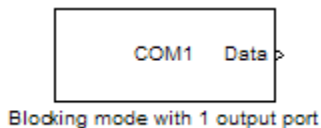
Instrument Control Toolbox

Description

The Serial Receive block configures and opens an interface to a specified remote address using the Serial protocol. The configuration and initialization occur once at the start of the model's execution. The block acquires data during the model's run time.

Note: You need a license for both the Instrument Control Toolbox and Simulink software to use this block.

This block has no input ports. It has one or two output ports based on your selection of blocking or nonblocking mode. If you select blocking mode, the block will have one output port corresponding to the data it receives.

Blocking mode with 1 output port

If you do not select blocking mode, the block will have two output ports, the **Data** port and the **Status** port.

24-13
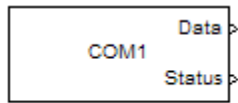
Non-blocking mode with 2 output ports

A First In, First Out (FIFO) buffer receives the data. At every time step, the **Data** port outputs the requested values from the buffer. In a nonblocking mode, the **Status** port indicates if the block has received new data. If the **Status** port displays 1 it means new data is available and if the **Status** port indicates 0 it means no new data is available.

Dialog Box

Use the Source Block Parameters dialog box to select your serial Receive block configuration options.

Note: Configure your serial port parameters before you specify the source block parameters.
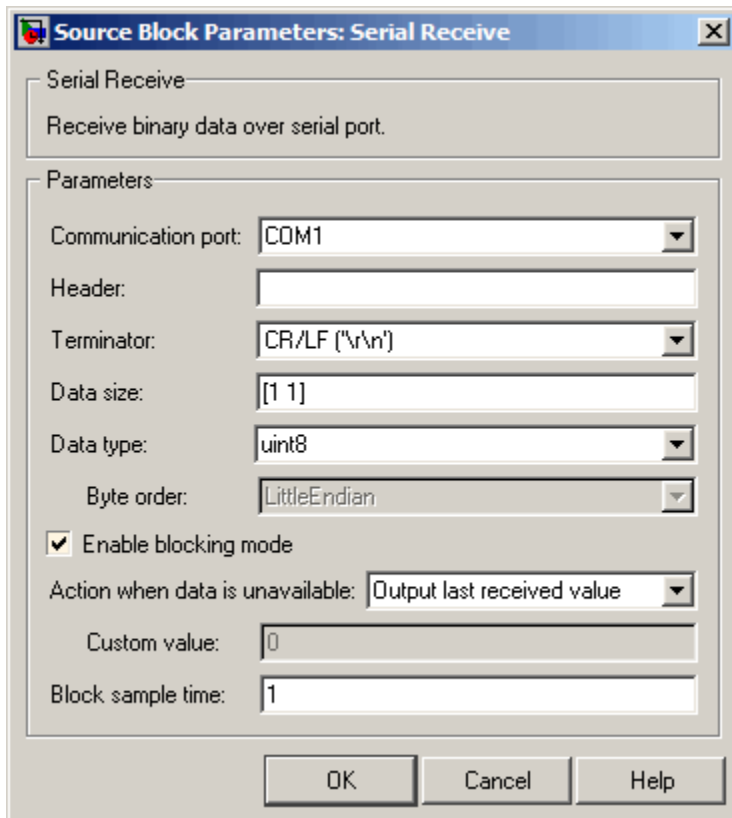
Communication port

Specify the serial port that you will use to receive from. You have to select an available port from the list. By default, the **Communication port** field contains the text **Please select a port...** and you must change this to a valid port. If you have not configured a port, the block will prompt you to do so. You can select a port from the available ports and then configure the port using the Serial Configuration block. Each Serial Receive block must have a configured serial port. If you use multiple ports in your simulation, you must configure each port separately.

Header

Specify data that marks the beginning of your data block. The header indicates the beginning of a new data block and the simulation will disregard data that occurs before the header. The header data is not sent to the output port. Only data that

occurs between the header and the terminator is sent to the output port. By default none or no header is specified.

Terminator

Specify data that marks the end of your data block. The terminator indicates the end of the data block and the simulation will account for any data that occurs after the terminator as a new data block. The terminator data is not sent to the output port. Only data that occurs between the header and the terminator is sent to the output port. By default <none> or no terminator is specified. Other available terminator formats are:

- CR ('\r') — Carriage return
- LF ('\n') — Line feed
- CR/LF ('\r\n')
- NULL ('\0')

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

Byte order

When you specify a data type other than int8 or uint8, you can specify the byte order of the device for the binary data. Your options are BigEndian or LittleEndian.

Enable blocking mode

Specify if you want to block the simulation while receiving data. This option is selected by default. Clear this check box if you do not want the read operation to block the simulation.

If you enable blocking mode, the model will block the simulation while it is waiting for the requested data to be available. 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.

Action when data is unavailable

Specify the action the block should take when data is unavailable. Available options are:

- **Output last received value** — Block will return the value it received at the preceding time step when it does not receive data at current time step. This value is selected by default.

- **Output custom value** — Block will return any user–defined value when it does not receive current data. You can define the custom value in the **Custom value** field.

- **Error** — Block will return an error when it does not receive current data. This option is unavailable if you do not select blocking mode.

Custom value

Specify a custom value for the block to output when it does not receive current data. The default value is 0. The custom value can be scalar or value equal to the size of Data that it receives (specified by **Data size** field).

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 1 second.

See Also

Query Instrument, Serial Configuration, Serial Send, TCP/IP Receive, TCP/IP Send, To Instrument, UDP Receive, UDP Send
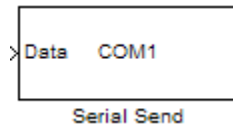
Serial Send

Send binary data over serial port

Library

Instrument Control Toolbox

Description

Serial Send

The Serial Send block sends binary data from your model to the specified remote machine using the serial protocol.


Note: You need a license for both the Instrument Control Toolbox and Simulink software to use this block.

The Serial 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. Valid data types are: single, double, int8, uint8, int16, uin16, int32, and uint32.

Dialog Box

Use the Sink Block Parameters dialog box to select your serial Send block configuration options.

Note: Configure your serial port parameters before you specify the source block parameters.
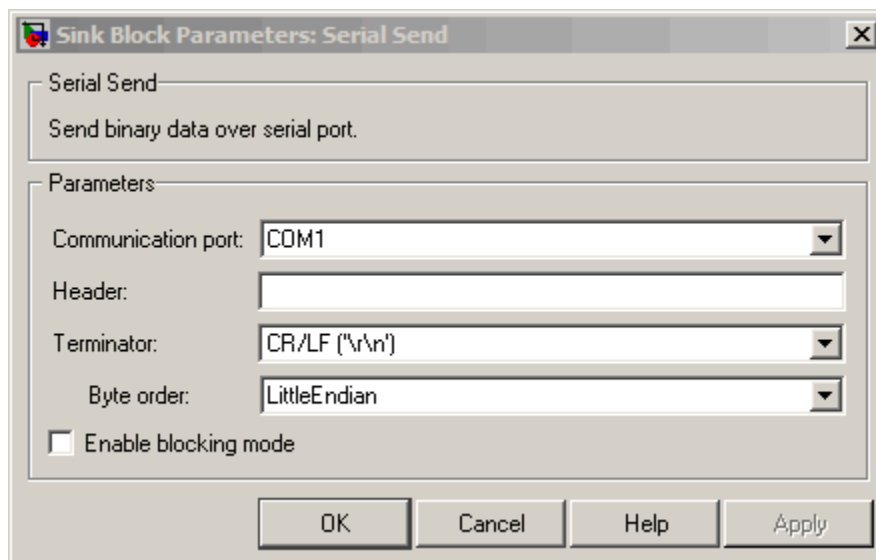
Communication port

Specify the serial port that you will use to send through. You have to select an available port from the list. If you have not configured a port, the block will prompt you to do so. You can select a port from the available ports and then configure the port using the Serial Configuration block. Each Serial Send block must have a configured serial port. If you use multiple ports in your simulation, you must configure each port separately.

Header

Specify supplemental data to be placed at the beginning of your data block. The Send block adds the header in front of the data before sending it over the serial port. By default none or no header is specified.

Terminator

Specify the supplemental data to be placed at the end of your data block. The Send blocks appends the terminator to the data before sending it over the serial port. By default `<none>` or no terminator is specified. Other available terminator formats are:

- `CR ('\r')` — Carriage return
- `LF ('\n')` — Line feed
- `CR/LF ('\r\n')`

- NULL (`'\0'`)

Byte order

When you specify a data type other than int8 or uint8, you can specify the byte order of the device for the binary data. Your options are `BigEndian` or `LittleEndian`.

Enable blocking mode

Specify if you want to block the simulation while sending data. This option is selected by default. Clear this check box if you do not want the write operation to block the simulation.

See Also

Query Instrument, `Serial Configuration`, `Serial Receive`, `TCP/IP Receive`, `TCP/IP Send`, `To Instrument`, `UDP Receive`, `UDP Send`

TCP/IP Receive

Receive data over TCP/IP from specified remote machine

Library

Instrument Control Toolbox


TCP/IP Receive

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 need a license for both the Instrument Control Toolbox and Simulink software to use this block.

This block has no input ports. It has either one or two output ports, based on your selection of blocking or nonblocking mode. If you select blocking mode then the block will have one output port corresponding to the data it receives.
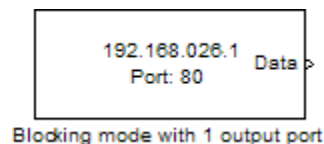
Blocking mode with 1 output port

If you do not select blocking mode, the block will have two output ports, the **Data** port and the **Status** port.
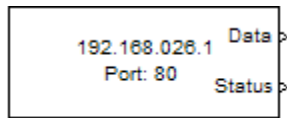
Non-blocking mode with 2 output ports

A First In First Out (FIFO) buffer receives the data. At every time step, the **Data** port outputs the requested values from the buffer. In nonblocking mode, the **Status** port indicates if the block has received new data.

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.
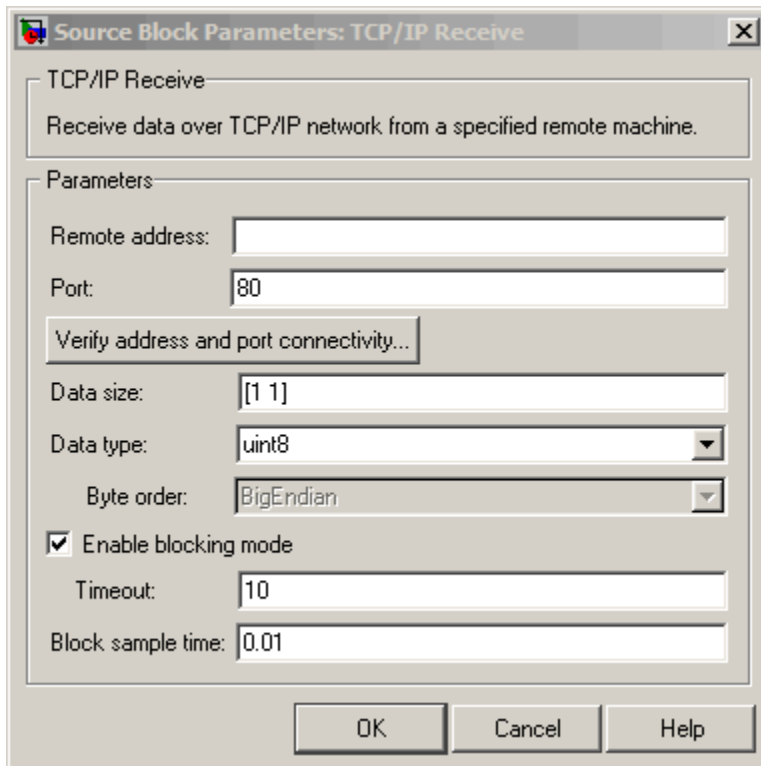
Remote address

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`

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`.
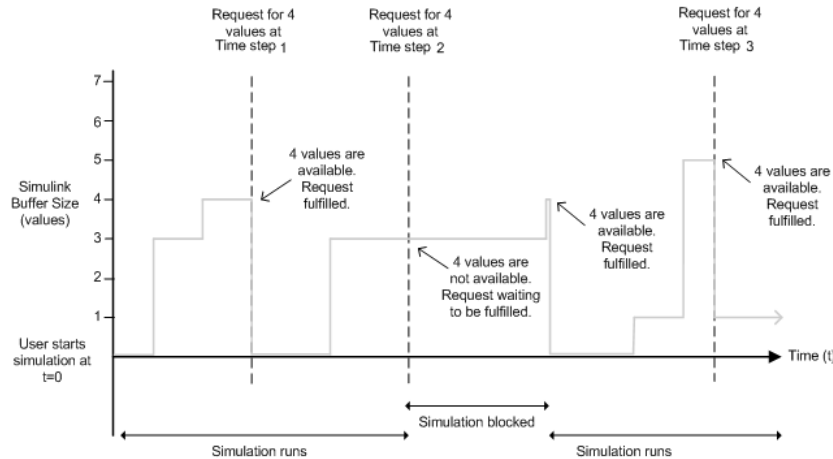
Enable blocking mode

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 specify 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.
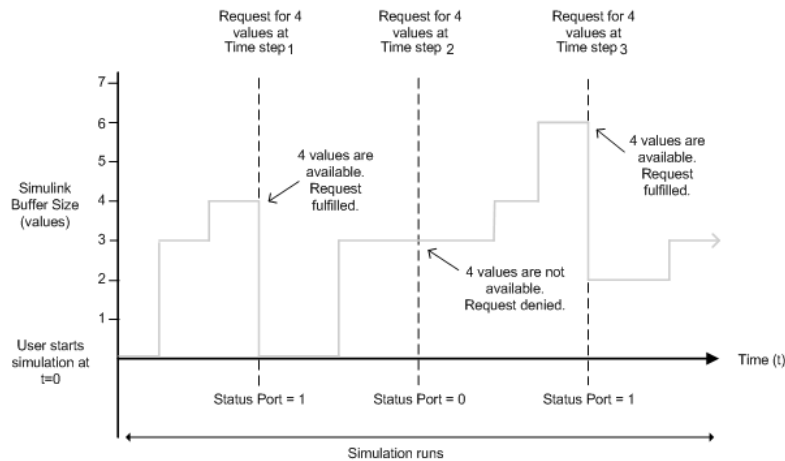
Blocking Mode

The blocking mode simulation occurs like this:

- At time step 1: The Simulink software requests data and the buffer has four values available. The block fulfills the request without interrupting the simulation. The block resets the buffer value to 0.

- At time step 2: The Simulink software requests data again, and the buffer has only three values, therefore it blocks the simulation until it receives the fourth value. When the block receives the fourth value, it fulfills the request and resumes the simulation. The block resets the buffer value to 0.

- At time step 3: When Simulink software requests data, the block has five values and it returns the first four that it received and resets the buffer to 1.

If the requested data is not received within the amount of time specified in the **Timeout** field (of the TCP/IP Receive Block Parameters dialog box), a Simulink error occurs and the simulation is stopped.

Note: In blocking mode, if you have more than one TCP/IP model on your computer, ensure that the Receive block is receiving data. If it is not, then your model might error out. You can avoid this by either changing the block to Nonblocking mode or by resetting the block's Priority.

Nonblocking Mode

Here the simulation is not blocked and runs continuously.

- At time step 1: The Simulink software requests data and the buffer has four values available, the block fulfills the request and changes the `Status` port value to `1`, indicating that new data is available. The `Data` port at this point contains the newly received values. The block resets the buffer value to 0.

- At time step 2: The Simulink software requests data again, and the buffer has only three values. The block cannot return a value of 3 because the data size is specified as 4. Therefore, the block sets the **Status** port value to `0`, indicating that there is no new data. The **Data** port contains the previously received value, and the buffer is at three (the number of values it received since the last request was fulfilled).

- At time step 3: When the Simulink software requests data here, the buffer now has five values and it returns the first four in the order 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 did not select **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).
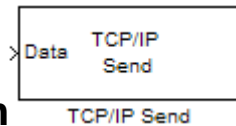
24-26

See Also

`Query Instrument`, `Serial Configuration`, `Serial Receive`, `Serial Send`, `TCP/IP Send`, `To Instrument`, `UDP Receive`, `UDP Send`

TCP/IP Send

Send data over TCP/IP to specified remote machine

Library

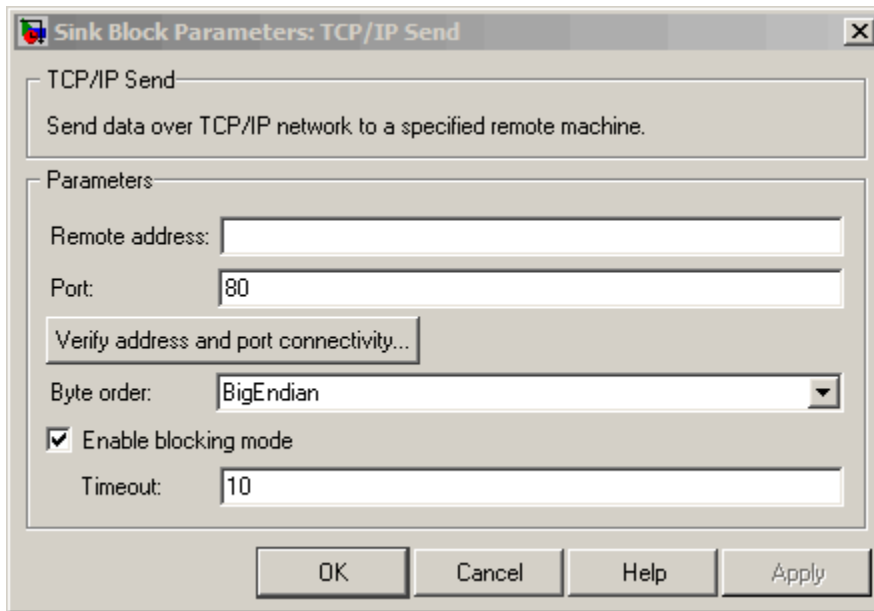Instrument Control Toolbox

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 need a license for both the Instrument Control Toolbox and Simulink software 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.

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.
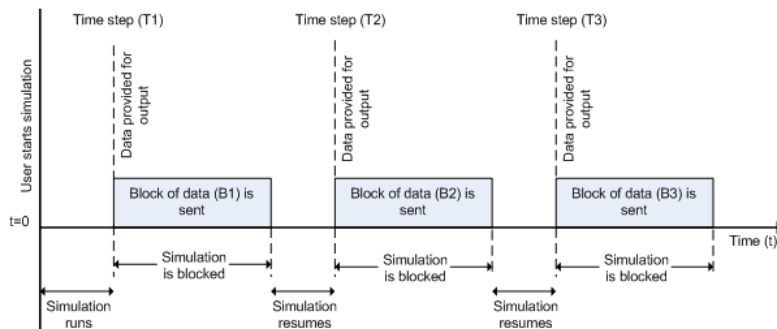- Establish connection with the specified remote address and port.

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.

Enable blocking mode

Specify if you want to block the simulation while sending data. This option is selected by default. Clear this check box if you do not want the write operation to block the simulation.
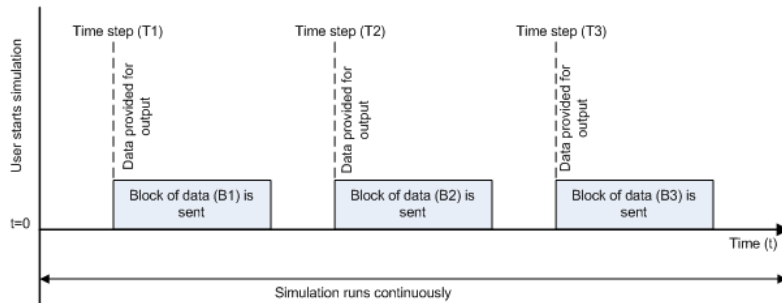
The following diagrams show the difference between sending data using blocking mode and nonblocking mode.
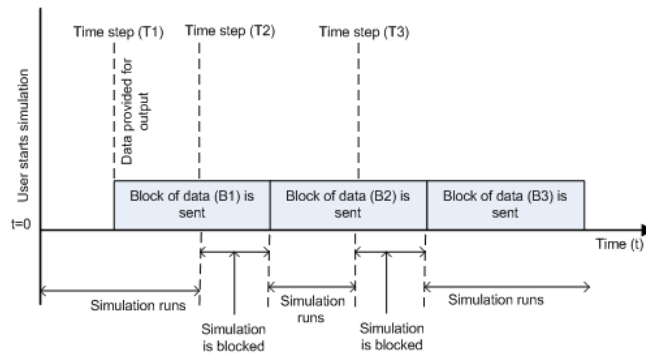
Blocking Mode

In this example, you start the simulation at time (t=0). At time step (T1), data output is initiated and simulation stops until the block of data (B1) is sent to the specified remote address and port. After the data is sent, simulation resumes until time step (T2), where the block initiates another data output and simulation is blocked until the block of data (B2) is sent to the remote address and port, and the simulation resumes.

Nonblocking Mode – Scenario 1

In this scenario, the data output outpaces the simulation speed. Data output is initiated at the first time step (T1) and the corresponding block of data (B1) is sent to the specified remote address asynchronously. The simulation runs continuously in this mode.

Nonblocking Mode – Scenario 2

In this scenario, the simulation 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 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.
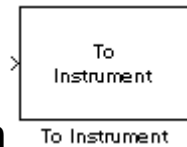
See Also

Query Instrument, Serial Configuration, Serial Receive, Serial Send, TCP/IP Receive, To Instrument, UDP Receive, UDP Send

To Instrument

Send simulation data to instrument

Library

Instrument Control Toolbox

Description

```
       ┌─────────────┐
       │     To      │
    ╲──│  Instrument │
       └─────────────┘
         To Instrument
```

The To Instrument block configures and opens an interface to an instrument, initializes the instrument, and sends data to the instrument. The configuration and initialization happen at the start of the model execution. The block sends data to the instrument during model run time.

The block has no output ports. The block has one input port corresponding to the data sent to the instrument. This data type must be double precision.

Note: The To Instrument block can be used with these interfaces: VISA, GPIB, Serial, TCP/IP, and UDP. It is not supported on these interfaces: SPI, I2C, and Bluetooth.

Dialog Box
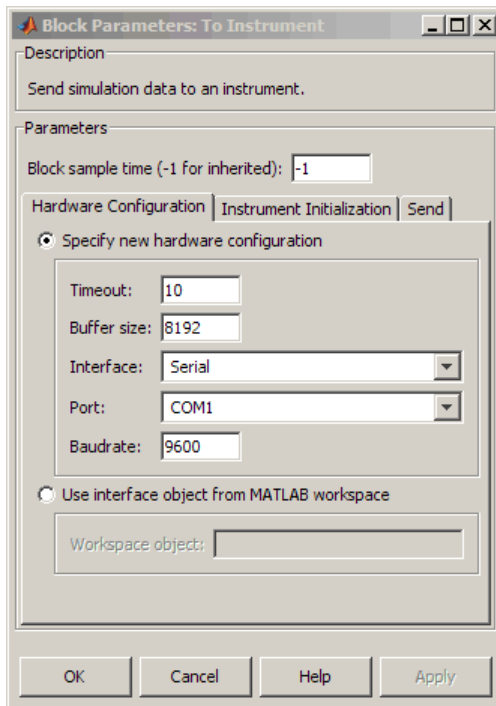
Block sample time

   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. See `BoardIndex` property for more information.

Board vendor

The vendor of the GPIB board used for the interface to the instrument. Your choices are Advantech, Agilent, Capital Equipment, Contec, ICS Electronics, 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 output buffer during a read operation.

Interface

Select the type of hardware interface to the instrument. Your options are those interfaces supported by the Instrument Control Toolbox software. The previous figure shows a configuration for a serial port interface.

Logical address

The logical address of the VXI instrument. Used for VISA VXI and VISA VXI-GPIB interface types.

Manufacturer ID

The manufacturer ID of the VISA USB instrument defined as a string. See `ManufacturerID` property for more information.

Model code

The model code of the VISA USB instrument defined as a string. See `ModelCode` property for more information.

Port

The port for the serial interface: COM1, COM2, etc.

Primary address

The primary address of the instrument on the GPIB.

Remote host

The host name or IP address of the instrument. Used for UDP, TCPIP, or VISA TCPIP interface types.

Remote port

The port on the instrument or remote host used for communication. Used for UDP, TCPIP, or VISA TCPIP interface types.

Secondary address

The secondary address of the instrument on the GPIB.

Serial number

> The serial number of the VISA USB instrument defined as a string. See `SerialNumber` property for more information.

Timeout

> Time in seconds, allowed to complete the query operation.

VISA vendor

> The vendor of the VISA instrument used for any of the VISA interface types. Your choices are Agilent, National Instruments, and Tektronix.

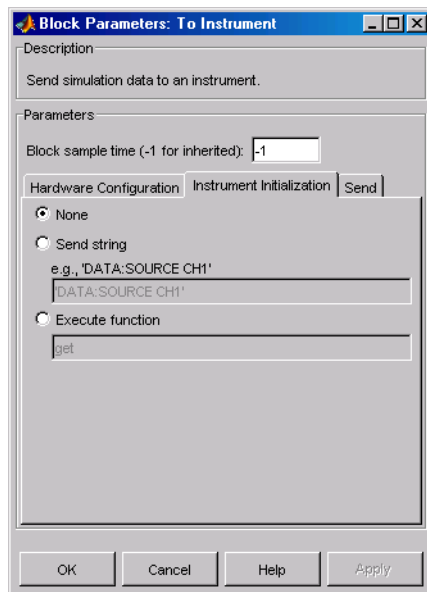**Use interface object from MATLAB workspace**

> Select this option to use an interface object from the MATLAB workspace.

Workspace object

> Enter the object name that you want to use from the MATLAB workspace.

Instrument Initialization Tab

The **Instrument Initialization** tab is where you define what happens when you first open your connection to the instrument.

None

> The default initialization option is none.

Send string

> A string sent to the instrument as an instrument command to initialize the instrument or set it up in a known state.

Execute function

> Any function that has as its only argument the interface object representing the instrument. You can write this function to include several instrument commands and initialization data.

Send Tab

The **Send** tab is where you define the optional command sent to the instrument and the format of the sent data.
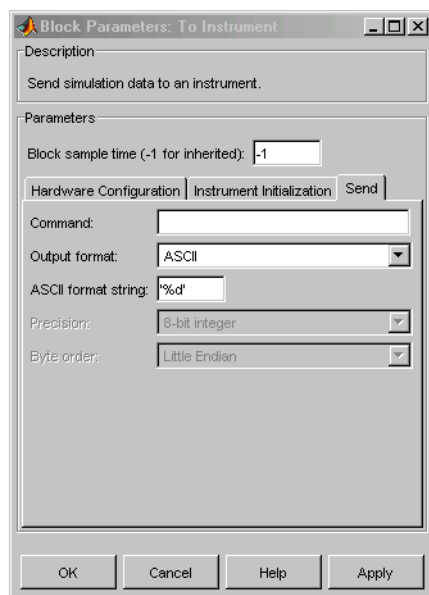
Command

> This is the command that is sent to the instrument with the Simulink data. *This command is optional*—if you leave this field blank, the Simulink data is sent to the instrument without any prefix or additional formatting.

Output format

Your options are ASCII, Binary, or Binblock (binary block — the binblock format is described in the binblockwrite function reference page).

ASCII format string

Available only when the format is ASCII, this defines the format string for the data. For a list of formats, see the fprintf function.

Precision

Used for binary or binblock format. Your options are:

- 8-bit integer (default)
- 16-bit integer
- 32-bit integer
- 8-bit unsigned integer
- 16-bit unsigned integer
- 32-bit unsigned integer
- 32-bit float
- 64-bit float

Byte order

When using binary or binblock format with more than 8 bits, you can specify the instrument's byte order for the data. Your options are Big Endian or Little Endian.

Tip Hardware information shown in the dialog box is determined and cached when you first open the dialog box. To refresh the display with new values, restart MATLAB.
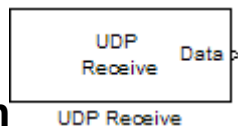

See Also

Query Instrument, TCP/IP Receive, TCP/IP Send, UDP Receive, UDP Send

UDP Receive

Receive data over UDP network from specified remote machine

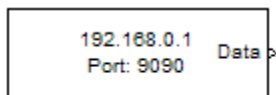## Library

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.
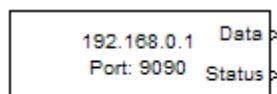

Note: You need a license for both the Instrument Control Toolbox and Simulink software to use this block.

This block has no input ports. It has either one or two output ports based on your selection of blocking or nonblocking mode. If you select blocking mode, the block will have one output port corresponding to the data it receives.

Blocking mode with 1 output port

If you do not select blocking mode, the block will have two output ports, the **Data** port and the **Status** port.

Non- blocking mode with 2 output ports

A First In First Out (FIFO) buffer receives the data. At every time step, the **Data** port outputs the requested values from the buffer. In a nonblocking mode, the **Status** port indicates if the block has received new data.

Dialog Box

Use the Source Block Parameters dialog box to select your acquisition mode and to set other configuration options.
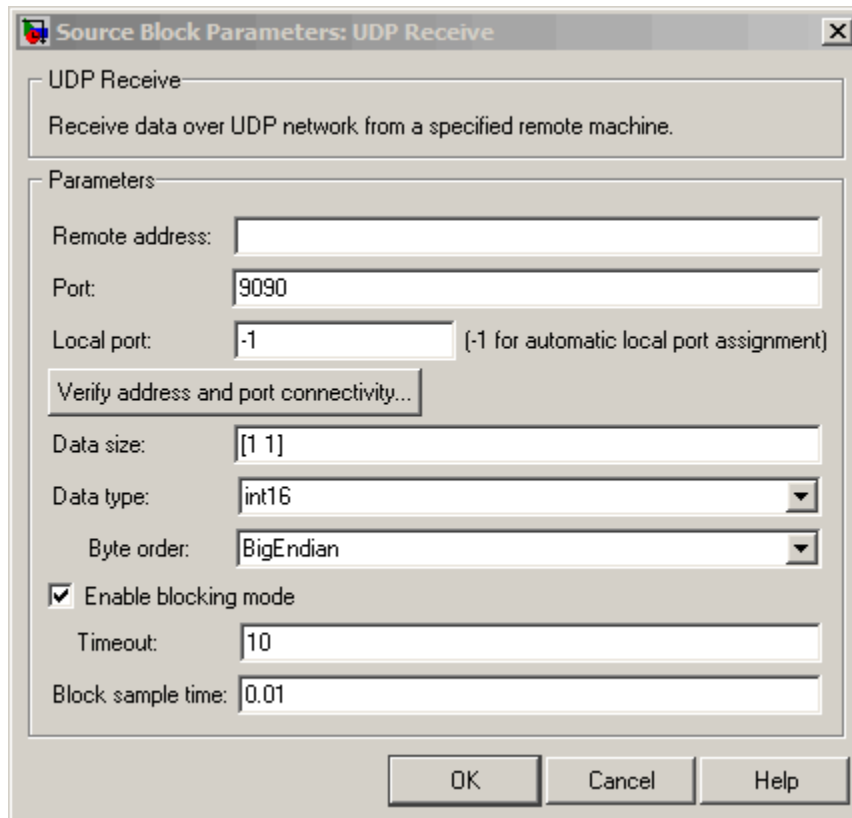
Remote address

> 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:

- 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

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.
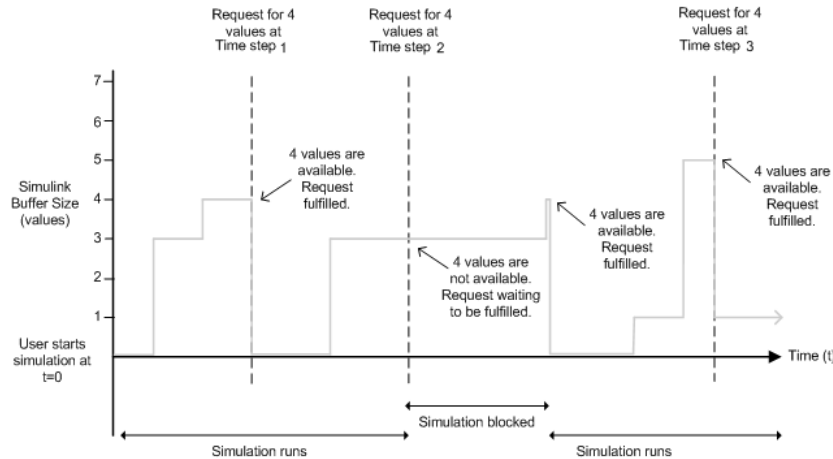
Enable blocking mode

Specify if you want to block the simulation while receiving data. This option is selected by default. Clear this check box if you do not want the read operation to block the simulation.

If you enable blocking mode, the model will block the simulation while it is waiting for the requested data to be available. If you do not enable blocking mode, the simulation runs continuously. The block has two output ports, **Status** and **Data**. The **Data** port contains the requested set of data at each time step. The **Status** port contains 0 or 1 based on whether it received new data at the given time step. The following diagrams show the difference between receiving data using blocking mode and nonblocking mode.

In this example, you start the simulation at time (t=0) and specify the amount of data to receive as 4 (set in the **Data size** field of the UDP Receive Block Parameters dialog box). After the simulation starts, the data is acquired asynchronously in a FIFO buffer.
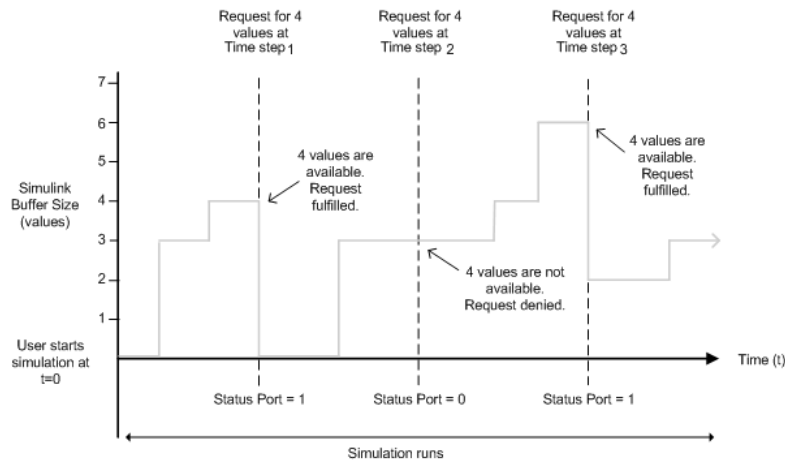
Blocking Mode

The blocking mode simulation occurs like this:

- At time step 1: The Simulink software requests data and the buffer has four values available, the block fulfills the request without interrupting the simulation. The block resets the buffer value to 0.
- At time step 2: The Simulink software requests data again, and the buffer has only three values, therefore it blocks the simulation until it receives the fourth value. When the block receives the fourth value, it fulfills the request and resumes the simulation. The block resets the buffer value to 0.
- At time step 3: When the Simulink software requests data, the block has five values and it returns the first four that it received and resets the buffer to 1.

If the requested data is not received within the amount of time specified in the **Timeout** field (of the UDP Receive Block Parameters dialog box), a Simulink software error occurs and the simulation is stopped.

Note: In blocking mode, if you have more than one UDP model on your computer, ensure that the Receive block is receiving data. If it is not, then your model might error out. You can avoid this by either changing the block to Nonblocking mode or by resetting the block's Priority.

Nonblocking Mode

Here the simulation is not blocked and runs continuously.

- At time step 1: The Simulink software requests data and the buffer has four values available, the block fulfills the request and changes the `Status` port value to `1`, indicating that new data is available. The `Data` port at this point contains the newly received values. The block resets the buffer value to 0.

- At time step 2: The Simulink software requests data again, and the buffer has only three values, and the block cannot return it as data size is specified as 4. Therefore the block sets the `Status` port value to `0`, indicating that there is no new data. The `Data` port contains the previously received value, and the buffer is at three (the number of values it has received since the last request was fulfilled).

- At time step 3: When the Simulink software requests data here, the buffer now has five values and it returns the first four in the order it received and changes the `Status` port value to `1`.

Timeout

Specify the amount of time that the model will wait for the data during each simulation time step. The default value is `inf` (seconds). This field is unavailable if you have not enabled blocking mode.

Block sample time

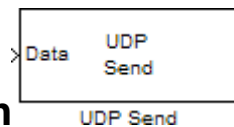Specify the sampling time of the block during simulation. The default value is `0.01` (seconds).

See Also

`Query Instrument`, `Serial Configuration`, `Serial Receive`, `Serial Send`, `TCP/IP Receive`, `TCP/IP Send`, `To Instrument`, `UDP Send`

UDP Send

Send data over UDP network to specified remote machine

Library

Instrument Control Toolbox

Description

The UDP Send block sends data from your model to the specified remote machine using the UDP protocol.

Note: You need a license for both the Instrument Control Toolbox and Simulink software to use this block.

The UDP Send block has one input port and it accepts both 1-D vector and matrix data. This block has no output ports. The block inherits the data type from the signal at the input port.

Dialog Box

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 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:

· Check if the specified remote address is correct.

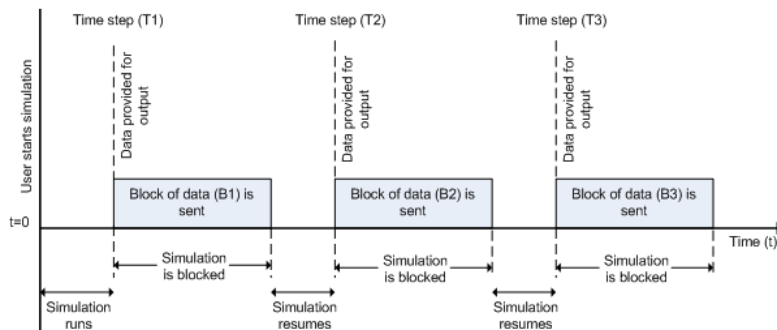· Establish connection with the specified remote address and port.

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`.

Enable blocking mode

Specify if you want to block the simulation while sending data. This option is selected by default. Clear this check box if you do not want the write operation to block the simulation.

The following diagrams show the difference between sending data using blocking mode and nonblocking mode.

Blocking Mode

In this example, you start the simulation at time (t=0). At time step (T1), data output is initiated and simulation stops until the block of data (B1) is sent to the specified remote address and port. After the data is sent, simulation resumes until time step (T2), where the block initiates another data output and simulation is blocked until the block of data (B2) is sent to the remote address and port, and the simulation resumes.

Nonblocking Mode – Scenario 1

In this scenario, the data output outpaces the simulation speed. Data output is initiated at the first time step (T1) and the corresponding block of data (B1) is sent to the specified remote address asynchronously. The simulation runs continuously in this mode.

Nonblocking Mode – Scenario 2

In this scenario, the simulation is nonblocking and occurs faster than the data initiation.

- At time step T1: The block of data (B1) is sent to the specified remote address and port asynchronously.
- At time step T2: The simulation is blocked until the block of data (B1) is sent completely. When the (B1) is completely sent, the new block of data (B2) is sent asynchronously, and the simulation resumes.

Note: Several factors, including network connectivity and model complexity, can affect the simulation speed. This can cause both nonblocking scenarios to occur within the same simulation.

See Also

Query Instrument, Serial Configuration, Serial Receive, Serial Send, TCP/IP Receive, TCP/IP Send, To Instrument, UDP Receive

Vendor Driver Requirements and Limitations

This appendix describes the requirements and limitations for the vendor GPIB and VISA drivers supported by the Instrument Control Toolbox software.

Note The limitations described in this appendix are restricted to the limitations directly associated with using the Instrument Control Toolbox software.

Driver Requirements

You can use the Instrument Control Toolbox software with the following GPIB and VISA drivers.

Interface	Vendor	Minimum Driver Requirements
GPIB	Advantech	Advantech version 435
	Agilent	Agilent IO Libraries version 15.0
	Capital Equipment Corporation	CEC Adaptor version 8.3
	CONTEC	Contec driver version 1.67
	ICS Electronics	ICS Adaptor version 2.0.13.0 ICS Adaptor version 3.2.0.0
	IOTech	IOTech Adaptor version 3.3
	Keithley	Keithley Adaptor version 1.10
	Measurement Computing Corporation	MCC Adaptor version 2.1 MCC Legacy Adaptor version 3.6
	National Instruments	NI Adaptor version 2.5.2
VISA	Agilent	Agilent IO Libraries version 15.0
	National Instruments	NI-VISA version 4.2.0
	Tektronix	Tektronix VISA version 3.3.0.14

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
"IOTech" 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
  - `BytesAvailableFcn` and `OutputEmptyFcn` 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
  - `BytesAvailableFcn` and `OutputEmptyFcn` 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 `BytesAvailableFcn` and `OutputEmptyFcn` 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 does not support asynchronous read and write operations, and therefore, ICS GPIB objects do not support the following toolbox functions and properties:

- `readasync`
- `async` flag for the `fprintf` and `fwrite`
- `BytesAvailableFcn`

- `OutputEmptyFcn`

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 `BytesAvailableFcn` and `OutputEmptyFcn` 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 `BytesAvailableFcn` and `OutputEmptyFcn` 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 `BytesAvailableFcn` and `OutputEmptyFcn` properties

VISA Driver Limitations

In this section...
"Agilent Technologies" on page A-7
"National Instruments" on page A-7

Agilent Technologies

The Agilent VISA driver uses all eight bits for the EOS comparison. Therefore, the only value that the `CompareBits` property supports 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`.

B

Bibliography

[1] Axelson, Jan, *Serial Port Complete*, Lakeview Research, Madison, WI, 1998.

[2] *Courier High Speed Modems User's Manual*, U.S. Robotics, Inc., Skokie, IL, 1994.

[3] TIA/EIA-232-F, *Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*.

[4] *Getting Started with Your AT Serial Hardware and Software for Windows 98/95*, National Instruments, Inc., Austin, TX, 1998.

[5] *HP E1432A User's Guide*, Hewlett-Packard Company, Palo Alto, CA, 1997.

[6] *HP 33120A Function Generator/Arbitrary Waveform Generator User's Guide*, Hewlett-Packard Company, Palo Alto, CA, 1997.

[7] *HP VISA User's Guide*, Hewlett-Packard Company, Palo Alto, CA, 1998.

[8] *NI-488.2M*™ *User Manual for Windows 95 and Windows NT*, National Instruments, Inc., Austin, TX, 1996.

[9] *NI-VISA*™ *User Manual*, National Instruments, Inc., Austin, TX, 1998.

[10] IEEE Std 488.2-1992, *IEEE Standard Codes, Formats, Protocols, and Common Commands for Use with IEEE Std 4881.-1987, IEEE Standard Digital Interface for Programmable Instrumentation*, Institute of Electrical and Electronics Engineers, New York, NY, 1992.

[11] *Instrument Communication Handbook*, IOTech, Inc., Cleveland, OH, 1991.

[12] *TDS 200-Series Two Channel Digital Oscilloscope Programmer Manual,* Tektronix, Inc., Wilsonville, OR.

[13] Stevens, W. Richard, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, Boston, MA, 1994.