Converting Data

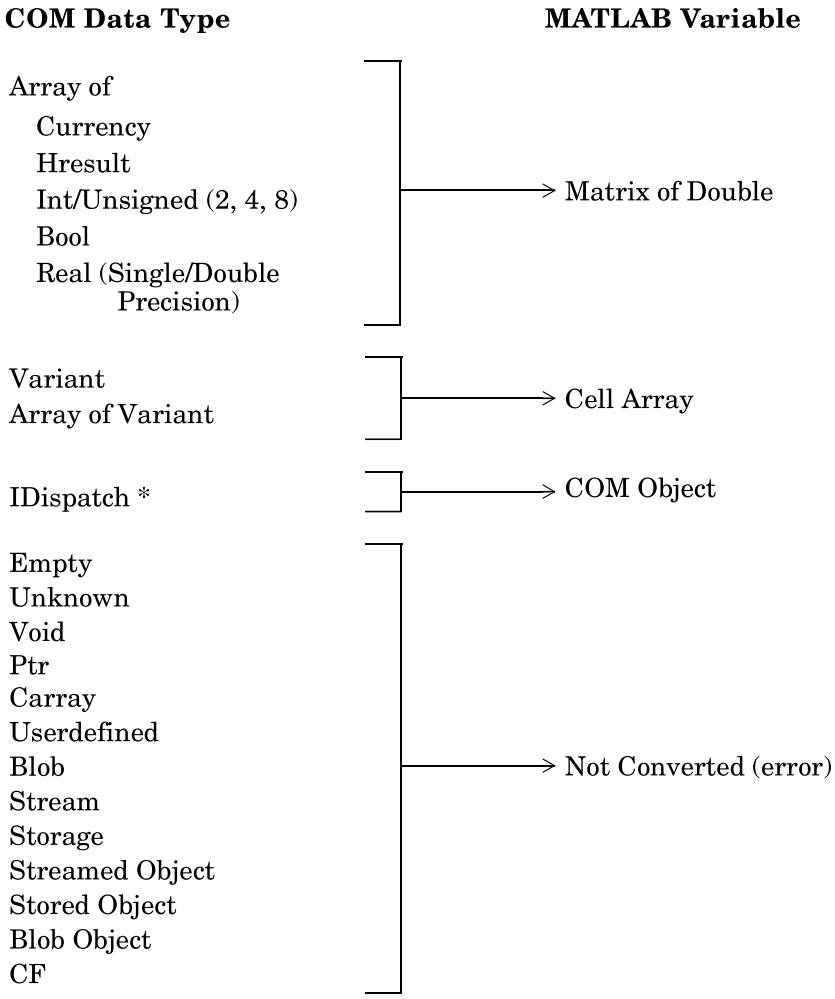
Since COM defines a number of different data formats and types, you will need to know how MATLAB converts data from COM objects into variables in the MATLAB workspace. Data from COM objects must be converted:

- When a property value is retrieved
- When a value is returned from a method invocation

This chart shows how COM data types are converted into variables in the MATLAB workspace.

COM Data Type	MATLAB Variable
String File Time Error Decimal Date	→ MATLAB String
Currency Hresult Int/Unsigned (2, 4, 8) Bool Real (Single/Double Precision)	→ Scalar Double
Null	→ NaN

Ex.1049 / Page 595 of 748
Intel Corporation et al



Using MATLAB as a DCOM Server Client

Distributed Component Object Model (DCOM) is an object distribution mechanism that allows clients to use remote COM objects over a network. Additionally, MATLAB can be used as a DCOM client with remote Automation

Ex.1049 / Page 596 of 748
Intel Corporation et al

servers if the operating system on which MATLAB is running is DCOM enabled.

Note If you use MATLAB as a remote DCOM server, all MATLAB windows will appear on the remote machine.

MATLAB COM Support Limitations

The following is a list of limitations of MATLAB COM support:

- MATLAB only supports indexed collections.
- COM controls are not printed with figure windows.
- MATLAB supports events from controls only, not from servers.
- MATLAB supports only the IDispatch interface. It does not support custom interfaces.

Ex.1049 / Page 597 of 748
Intel Corporation et al

MATLAB Automation Server Support

MATLAB on Microsoft Windows supports COM Automation server capabilities. Automation is a COM protocol that allows one application or component (the *controller*) to control another application or component (the *server*). Thus, a MATLAB server can be controlled by any Windows program that can be an Automation Controller. Some examples of applications that can be Automation Controllers are Microsoft Excel, Microsoft Access, Microsoft Project, and many Visual Basic and Visual C++ programs.

This section explains how to create and connect to an Automation server running MATLAB, how to call functions in the server from either MATLAB M-file or Visual Basic client applications, and how to use properties that affect the server.

- "Creating the Automation Server"
- "Automation Server Functions" on page 8-37
- "MATLAB Automation Properties" on page 8-39

Note If you plan to build your client application using either C or Fortran, we recommend that you use the MATLAB Engine facility instead of an Automation server.

Creating the Automation Server

Exactly how you create an Automation server depends on the controller you are using. Consult the documentation for your controller for this information. All controllers require a Program ID (ProgID) to identify the server. The ProgID registered for MATLAB is `matlab.application`.

If your controller is a MATLAB application, you can create the Automation server using the MATLAB `actxserver` function:

```
h = actxserver('matlab.application')
h =
   COM.matlab.application
```

Usually, the Automation server is automatically created by Windows when the controller first establishes its connection to the server. You may also choose to create the server manually. See “Creating the Server Manually” on page 8-40.

Shared and Dedicated Servers

You can start a MATLAB Automation server in either of two modes:

- Shared — One or more client applications connect to the same MATLAB server. The server is shared between all clients.
- Dedicated — Each client application creates its own dedicated MATLAB server.

If you use `matlab.application` as your ProgID, MATLAB creates a shared server. See “Specifying a Shared or Dedicated Server” on page 8-40.

Startup Directory

The Automation server starts up in the directory displayed by typing the following at the MATLAB prompt:

```
[matlabroot '\bin\win32']
```

If this is not your usual MATLAB startup directory, then the newly created server may not have access to files in that directory and also will not run your MATLAB startup file (`startup.m`) when the server process starts MATLAB. To give the server access files in your startup directory, you need to either have the server `cd` to that directory, add it to the server’s MATLAB path (using `addpath`), or include the pathname when referencing those files.

Automation Server Functions

MATLAB provides a number of functions to enable an Automation controller written in either MATLAB or Visual Basic to manipulate data in the MATLAB server. These are shown in the table below and are described in individual function reference pages.

Ex.1049 / Page 599 of 748
Intel Corporation et al

For a description of how to use these functions in controller programs written in the MATLAB programming language, see “COM Server Functions for the MATLAB Client”. For controller programs written in Viaual Basic, see “COM Server Functions for the Visual Basic Client”.

Function	Description
`Execute`	Execute MATLAB command in server
`GetCharArray`	Get character array from server
`GetFullMatrix`	Get matrix from server
`MaximizeCommandWindow`	Display server window on Windows desktop
`MinimizeCommandWindow`	Minimize size of server window
`PutCharArray`	Store character array in server
`PutFullMatrix`	Store matrix in server
`Quit`	Terminate MATLAB server

For MATLAB Clients

To see a summary of all functions available to controller applications along with the required syntax, start a MATLAB Automation server, and then use the `invoke` function with only the `handle` argument:

```
handle = actxserver('matlab.application');
invoke(handle)
```

For Visual Basic Clients

Data types for the arguments and return values of the server functions are expressed as Automation data types, which are language-independent types defined by the Automation protocol. For example, `BSTR` is a wide-character string type defined as an Automation type, and is the same data format used by Visual Basic to store strings. Any COM-compliant controller should support these data types, although the details of how you declare and manipulate these are controller specific.

MATLAB Automation Properties

You have the option of making your server application visible or not by setting the `Visible` property. When visible, the server window appears on the desktop, enabling the user to interact with the server application. This may be useful for such purposes as debugging. When not visible, the server window does not appear, thus perhaps making for a cleaner interface and also preventing any interaction with the server application.

By default, the `Visible` property is enabled, or set to `1`:

```
h = actxserver('Matlab.Application');
get(h, 'visible')
ans =
     1
```

You can change the setting of `Visible` by setting it to `0` (invisible) or `1` (visible). The following command removes the server application window from the desktop:

```
set(h, 'visible', 0);

get(h, 'visible')
ans =
     0
```


Ex.1049 / Page 601 of 748
Intel Corporation et al

Additional Automation Server Information

This section covers several other topics related to Automation Servers:

- "Creating the Server Manually"
- "Specifying a Shared or Dedicated Server"
- "Using MATLAB as a DCOM Server" on page 8-41

Creating the Server Manually

The Automation server is created automatically by Windows when a controller application first establishes a server connection. Alternatively, you may choose to create the server manually, prior to starting any of the client processes.

To do this, use the `/Automation` switch in the MATLAB startup command:

1. Right-click the MATLAB shortcut icon and select **Properties** from the context menu. The **Properties** dialog box for `matlab.exe` opens to the **Shortcut** panel.
2. In the **Target** field, after the target path for `matlab.exe`, add `/Automation`.

Note When Windows automatically creates a MATLAB server, it too uses the `/Automation` switch. In this way, MATLAB servers are differentiated from other MATLAB sessions. This protects controllers from interfering with any interactive MATLAB sessions that may be running.

Specifying a Shared or Dedicated Server

You can start the MATLAB Automation server in one of two modes — shared or dedicated. A dedicated server is dedicated to a single client; a shared server is shared by multiple clients. The mode is determined by the Program ID (ProgID) used by the client to start MATLAB.

Ex.1049 / Page 602 of 748
Intel Corporation et al

Starting a Shared Server

The ProgID, `matlab.application`, specifies the default mode, which is shared. You can also use the version-specific ProgID, `matlab.application.N`, where `N` is equal to the major version of MATLAB you are running, (for example, `N = 6` for MATLAB 6.5).

Once MATLAB is started as a shared server, all clients that request a connection to MATLAB by using the shared server ProgID connect to the already running instance of MATLAB. In other words, there is never more than one instance of a shared server running, since it is shared by all clients that use the ProgID that specifies a shared server.

Starting a Dedicated Server

To specify a dedicated server, use the ProgID, `matlab.application.single`, (or the version-specific ProgID, `matlab.application.single.N`).

Each client that requests a connection to MATLAB using a dedicated ProgID creates a separate instance of MATLAB, and that server will not be shared with any other client. Therefore, there can be several instances of a dedicated server running simultaneously, since the dedicated server is not shared by multiple clients.

Using MATLAB as a DCOM Server

Distributed Component Object Model (DCOM) is a protocol that allows COM connections to be established over a network. If you are using a version of Windows that supports DCOM (Windows NT 4.0 at the time of this writing) and a controller that supports DCOM, you can use the controller to start a MATLAB server on a remote machine.

To do this, DCOM must be configured properly, and MATLAB must be installed on each machine that is used as a client or server. (Even though the client machine may not be running MATLAB in such a configuration, the client machine must have a MATLAB installation because certain MATLAB components are required to establish the remote connection.) Consult the DCOM documentation for how to configure DCOM for your environment.

Ex.1049 / Page 603 of 748
Intel Corporation et al

Dynamic Data Exchange (DDE)

MATLAB provides functions that enable MATLAB to access other Windows applications and for other Windows applications to access MATLAB in a wide range of contexts. These functions use dynamic data exchange (DDE), software that allows Microsoft Windows applications to communicate with each other by exchanging data.

This section describes using DDE in MATLAB:

- "DDE Concepts and Terminology" on page 8-42
- "Accessing MATLAB as a Server" on page 8-44
- "The DDE Name Hierarchy" on page 8-45
- "Example: Using Visual Basic and the MATLAB DDE Server" on page 8-48
- "Using MATLAB as a Client" on page 8-50
- "DDE Advisory Links" on page 8-51

DDE Concepts and Terminology

Applications communicate with each other by establishing a DDE *conversation*. The application that initiates the conversation is called the *client*. The application that responds to the client application is called the *server*.

When a client application initiates a DDE conversation, it must identify two DDE parameters that are defined by the server:

- The name of the application it intends to have the conversation with, called the *service name*
- The subject of the conversation, called the *topic*

When a server application receives a request for a conversation involving a supported topic, it acknowledges the request, establishing a DDE conversation. The combination of a service and a topic identifies a conversation uniquely. The service or topic cannot be changed for the duration of the conversation, although the service can maintain more than one conversation.

During a DDE conversation, the client and server applications exchange data concerning items. An *item* is a reference to data that is meaningful to both applications in a conversation. Either application can change the item during a conversation. These concepts are discussed in more detail below.

Ex.1049 / Page 604 of 748
Intel Corporation et al

The Service Name

Every application that can be a DDE server has a unique *service name*. The service name is usually the application's executable filename without any extension. Service names are not case sensitive. Here are some commonly used service names:

- The service name for MATLAB is *Matlab*.
- The service name for Microsoft Word for Windows is *WinWord*.
- The service name for Microsoft Excel is *Excel*.

For the service names of other Windows applications, refer to the application documentation.

The Topic

The *topic* defines the subject of a DDE conversation and is usually meaningful to both the client and server applications. Topic names are not case sensitive. MATLAB topics are *System* and *Engine* and are discussed in "Accessing MATLAB as a Server" on page 8-44. Most applications support the *System* topic and at least one other topic. Consult your application documentation for information about supported topics.

The Item

Each topic supports one or more items. An *item* identifies the data being passed during the DDE conversation. Case sensitivity of items depends on the application. MATLAB *Engine* items are case sensitive if they refer to matrices because matrix names are case sensitive.

Clipboard Formats

DDE uses the Windows clipboard formats for formatting data sent between applications. As a client, MATLAB supports only Text format. As a server, MATLAB supports Text, Metafilepict, and XLTable formats, described below:

- **Text** – Data in Text format is a buffer of characters terminated by the `null` character. Lines of text in the buffer are delimited by a carriage return line-feed combination. If the buffer contains columns of data, those columns are delimited by the tab character. MATLAB supports Text format for obtaining the results of a remote EvalString command and requests for matrix data. Also, matrix data can be sent to MATLAB in Text format.

Ex.1049 / Page 605 of 748
Intel Corporation et al

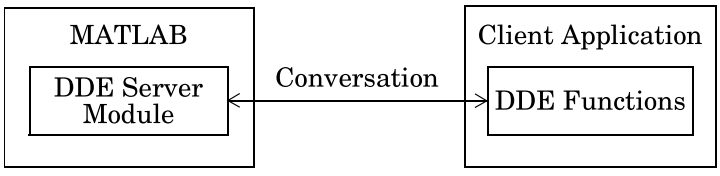
- **Metafilepict** – Metafilepict format is a description of graphical data containing the drawing commands for graphics. As a result, data stored in this format is scalable and device independent. MATLAB supports Metafilepict format for obtaining the result of a remote command that causes some graphic action to occur.
- **XLTable** – XLTable format is the clipboard format used by Microsoft Excel and is supported for ease and efficiency in exchanging data with Excel. XLTable format is a binary buffer with a header that describes the data held in the buffer. For a full description of XLTable format, consult the Microsoft Excel SDK documentation.

Accessing MATLAB as a Server

A client application can access MATLAB as a DDE server in the following ways, depending on the client application:

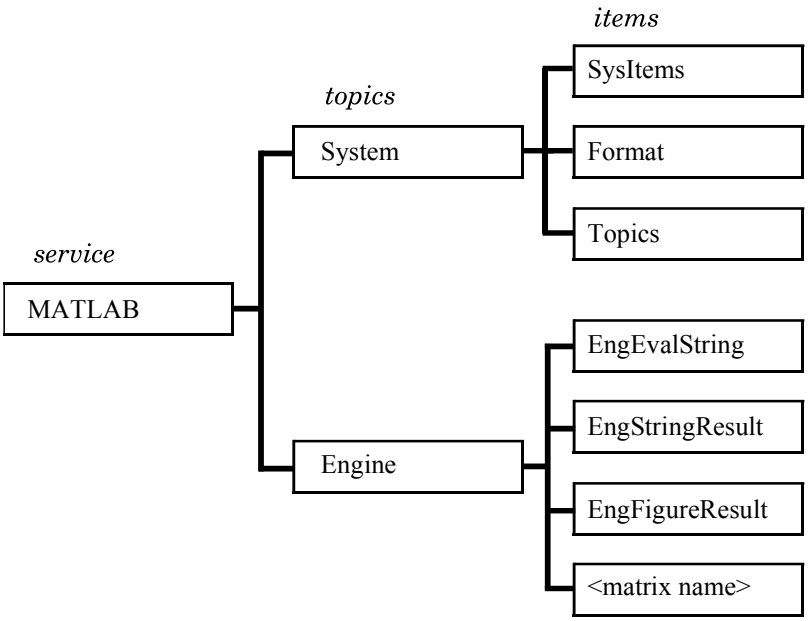
- If you are using an application that provides functions or macros to conduct DDE conversations, you can use these functions or macros. For example, Microsoft Excel, Word for Windows, and Visual Basic provide DDE functions or macros. For more information about using these functions or macros, see the appropriate Microsoft documentation.
- If you are creating your own application, you can use the MATLAB Engine Library or DDE directly. For more information about using the Engine Library, see "Using the MATLAB Engine" on page 6-2. For more information about using DDE routines, see the *Microsoft Windows Programmer's Guide*.

The figure below illustrates how MATLAB communicates as a server. DDE functions in the client application communicate with the MATLAB DDE server module. The client's DDE functions can be provided by either the application or the MATLAB Engine Library.



The DDE Name Hierarchy

When you access MATLAB as a server, you must specify its service name, topic, and item. The figure below illustrates the MATLAB DDE name hierarchy. Topics and items are described in more detail below.



The two MATLAB topics are *System* and *Engine*.

MATLAB System Topic

The System topic allows users to browse the list of topics provided by the server, the list of System topic items provided by the server, and the formats supported by the server.

The MATLAB System topic supports these items:

- **SysItems**

  Provides a tab-delimited list of items supported under the System topic (this list).

Ex.1049 / Page 607 of 748
Intel Corporation et al

- **Format**

  Provides a tab-delimited list of string names of all the formats supported by the server. MATLAB supports Text, Metafilepict, and XLTable. These formats are described in “Clipboard Formats” on page 8-43.

- **Topics**

  Provides a tab-delimited list of the names of the topics supported by MATLAB.

MATLAB Engine Topic

The Engine topic allows users to use MATLAB as a server by passing it a command to execute, requesting data, or sending data.

The MATLAB Engine topic supports these items:

- **EngEvalString**

  Specifies an item name, if required, when you send a command to MATLAB for evaluation.

- **EngStringResult**

  Provides the string result of a DDE execute command when you request data from MATLAB.

- **EngFigureResult**

  Provides the graphical result of a DDE execute command when you request data from MATLAB.

- **<matrix name>**

  When requesting data from MATLAB, this is the name of the matrix for which data is being requested. When sending data to MATLAB, this is the name of the matrix to be created or updated.

The MATLAB Engine topic supports three operations that may be used by applications with a DDE client interface. These operations include sending commands to MATLAB for evaluation, requesting data from MATLAB, and sending data to MATLAB.

Sending Commands to MATLAB for Evaluation

Clients send commands to MATLAB using the DDE `execute` operation. The Engine topic supports DDE `execute` in two forms because some clients require that you specify the item name and the command to execute, while others

Ex.1049 / Page 608 of 748
Intel Corporation et al

require only the command. Where an item name is required, use `EngEvalString`. In both forms, the format of the command must be Text. Most clients default to Text for DDE `execute`. If the format cannot be specified, it is probably Text. The table summarizes the DDE `execute` parameters.

Item	Format	Command
`EngEvalString`	Text	String
`null`	Text	String

Requesting Data from MATLAB

Clients request data from MATLAB using the DDE `request` operation. The Engine topic supports DDE requests for three functions:

Text that is the result of the previous DDE execute command. You request the string result of a DDE execute command using the `EngStringResult` item with Text format.

Graphical results of the previous DDE execute command. You request the graphical result of a DDE execute command using the `EngFigureResult` item. The `EngFigureResult` item can be used with Text or Metafilepict formats:

- Specifying the Text format results in a string having a value of “yes” or “no.” If the result is “yes,” the metafile for the current figure is placed on the clipboard. This functionality is provided for DDE clients that can retrieve only text from DDE requests, such as Word for Windows. If the result is “no,” no metafile is placed on the clipboard.
- Specifying the Metafilepict format when there is a graphical result causes a metafile to be returned directly from the DDE request.

The data for a specified matrix. You request the data for a matrix by specifying the name of the matrix as the item. You can specify either the Text or XLTable format.

The table summarizes the DDE `request` parameters.

Item	Format	Result
`EngStringResult`	Text	String
`EngFigureResult`	Text	Yes/No
`EngFigureResult`	Metafilepict	Metafile of the current figure
`<matrix name>`	Text	Character buffer, tab-delimited columns, CR/LF-delimited rows
`<matrix name>`	XLTable	Binary data in a format compatible with Microsoft Excel

Sending Data to MATLAB

Clients send data to MATLAB using the DDE `poke` operation. The Engine topic supports DDE poke for updating or creating new matrices in the MATLAB workspace. The item specified is the name of the matrix to be updated or created. If a matrix with the specified name already exists in the workspace it will be updated; otherwise it will be created. The matrix data can be in Text or XLTable format.

The table summarizes the DDE `poke` parameters.

Item	Format	Poke Data
`<matrix name>`	Text	Character buffer, tab-delimited columns, CR/LF-delimited rows
`<matrix name>`	XLTable	Binary data in a format compatible with Microsoft Excel

Example: Using Visual Basic and the MATLAB DDE Server

This example shows a Visual Basic form that contains two text edit controls, **TextInput** and **TextOutput**. This code is the `TextInput_KeyPress` method.

```
Sub TextInput_KeyPress(KeyAscii As Integer)
rem If the user presses the return key
rem in the TextInput control.
If KeyAscii = vbKeyReturn then

rem Initiate the conversation between the TextInput
rem control and MATLAB under the Engine topic.
rem Set the item to EngEvalString.
                  TextInput.LinkMode = vbLinkNone
                  TextInput.LinkTopic = "MATLAB|Engine"
                  TextInput.LinkItem = "EngEvalString"
                  TextInput.LinkMode = vbLinkManual

rem Get the current string in the TextInput control.
rem This text is the command string to send to MATLAB.
                  szCommand = TextInput.Text

rem Perform DDE Execute with the command string.
                  TextInput.LinkExecute szCommand
                  TextInput.LinkMode = vbLinkNone

rem Initiate the conversation between the TextOutput
rem control and MATLAB under the Engine topic.
rem Set the item to EngStringResult.
                  TextOutput.LinkMode = vbLinkNone
                  TextOutput.LinkTopic = "MATLAB|Engine"
                  TextOutput.LinkItem = "EngStringResult"
                  TextOutput.LinkMode = vbLinkManual

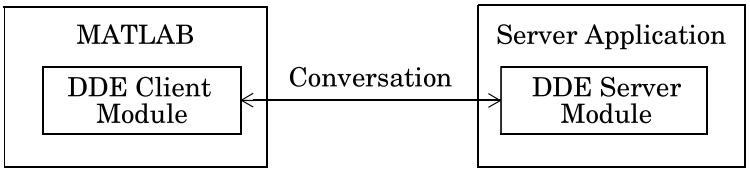
rem Request the string result of the previous EngEvalString
rem command. The string ends up in the text field of the
rem control TextOutput.text.
                  TextOutput.LinkRequest
                  TextOutput.LinkMode = vbLinkNone

End If
End Sub
```

Using MATLAB as a Client

For MATLAB to act as a client application, you can use the MATLAB DDE client functions to establish and maintain conversations.

This figure illustrates how MATLAB communicates as a client to a server application.



The MATLAB DDE client module includes a set of functions. This table describes the functions that enable you to use MATLAB as a client.

Function	Description
`ddeadv`	Sets up advisory link between MATLAB and DDE server application.
`ddeexec`	Sends execution string to DDE server application.
`ddeinit`	Initiates DDE conversation between MATLAB and another application.
`ddepoke`	Sends data from MATLAB to DDE server application.
`ddereq`	Requests data from DDE server application.
`ddeterm`	Terminates DDE conversation between MATLAB and server application.
`ddeunadv`	Releases advisory link between MATLAB and DDE server application.

If the server application is Microsoft Excel, you can specify the *System* topic or a topic that is a filename. If you specify the latter, the filename ends in `.XLS` or

Ex.1049 / Page 612 of 748
Intel Corporation et al

`.XLC` and includes the full path if necessary. A Microsoft Excel item is a cell reference, which can be an individual cell or a range of cells.

Microsoft Word for Windows topics are *System* and document names that are stored in files whose names end in `.DOC` or `.DOT`. A Word for Windows item is any bookmark in the document specified by the topic.

The following example is an M-file that establishes a DDE conversation with Microsoft Excel, and then passes a 20-by-20 matrix of data to Excel:

```
% Initialize conversation with Excel.
chan = ddeinit('excel', 'Sheet1');

% Create a surface of peaks plot.
h = surf(peaks(20));
% Get the z data of the surface
z = get(h, 'zdata');

% Set range of cells in Excel for poking.
range = 'r1c1:r20c20';

% Poke the z data to the Excel spread sheet.
rc = ddepoke(chan, range, z);
```

DDE Advisory Links

You can use DDE to notify a client application when data at a server has changed. For example, if you use MATLAB to analyze data entered in an Excel spreadsheet, you can establish a link that causes Excel to notify MATLAB when this data changes. You can also establish a link that automatically updates a matrix with the new or modified spreadsheet data.

MATLAB supports two kinds of advisory links, distinguished by the way in which the server application advises MATLAB when the data that is the subject of the item changes at the server:

- A *hot link* causes the server to supply the data to MATLAB when the data defined by the item changes.
- A *warm link* causes the server to notify MATLAB when the data changes but supplies the data only when MATLAB requests it.

You set up and release advisory links with the `ddeadv` and `ddeunadv` functions. MATLAB only supports links when MATLAB is a client.

This example establishes a DDE conversation between MATLAB, acting as a client, and Microsoft Excel. The example extends the example in the previous section by creating a hot link with Excel. The link updates matrix `z` and evaluates a callback when the range of cells changes. A push-button, user interface control terminates the advisory link and the DDE conversation when pressed. (For more information about creating a graphical user interface, see the online MATLAB manual, *Creating Graphical User Interfaces*.)

```
% Initialize conversation with Excel.
chan = ddeinit('excel', 'Sheet1');

% Set range of cells in Excel for poking.
range = 'r1c1:r20c20';

% Create a surface of peaks plot.
h = surf(peaks(20));

% Get the z data of the surface.
z = get(h, 'zdata');

% Poke the z data to the Excel spread sheet.
rc = ddepoke(chan, range, z);

% Set up a hot link ADVISE loop with Excel
% and the MATLAB matrix 'z'.
% The callback sets the zdata and cdata for
% the surface h to be the new data sent from Excel.
rc = ddeadv(chan, range,...
     'set(h,''zdata'',z);set(h,''cdata'',z);','z');

% Create a push button that will end the ADVISE link,
% terminate the DDE conversation,
% and close the figure window.
c = uicontrol('String','&Close','Position',[5 5 80 30],...
     'Callback',...
     'rc = ddeunadv(chan,range);ddeterm(chan);close;');
```

9

Serial Port I/O

The serial port I/O chapter includes these sections.

Introduction (p. 9-2)	Serial port capabilities, supported interfaces, and supported platforms.
Overview of the Serial Port (p. 9-4)	The serial port interface standard, signals and pin assignments, the serial data format, and finding serial port information for your platform.
Getting Started with Serial I/O (p. 9-18)	Examples to help you get started with the serial port interface.
Creating a Serial Port Object (p. 9-24)	Create a MATLAB object that represents the serial I/O device.
Connecting to the Device (p. 9-27)	Establish a connection between MATLAB and the serial I/O device.
Configuring Communication Settings (p. 9-28)	Set values for the baud rate and the serial data format.
Writing and Reading Data (p. 9-29)	Write data to the device and read data from the device.
Events and Callbacks (p. 9-48)	Enhance your serial I/O application by using events and callbacks.
Using Control Pins (p. 9-56)	Signal the presence of connected devices and control the flow of data.
Debugging: Recording Information to Disk (p. 9-62)	Save transferred data and event information to disk.
Saving and Loading (p. 9-68)	Save and load serial port objects.
Disconnecting and Cleaning Up (p. 9-69)	Disconnect the serial port object from the device, and remove the object from memory and from the workspace.
Property Reference (p. 9-70)	Properties grouped by category.

Introduction

What Is the MATLAB Serial Port Interface?

The MATLAB serial port interface provides direct access to peripheral devices such as modems, printers, and scientific instruments that you connect to your computer's serial port. This interface is established through a serial port object. The serial port object supports functions and properties that allow you to

- Configure serial port communications
- Use serial port control pins
- Write and read data
- Use events and callbacks
- Record information to disk

If you want to communicate with PC-compatible data acquisition hardware such as multifunction I/O boards, you need the Data Acquisition Toolbox. If you want to communicate with GPIB- or VISA-compatible instruments, you need the Instrument Control Toolbox. Note that this toolbox also includes additional serial I/O utility functions that facilitate object creation and configuration, instrument communication, and so on.

For more information about these products, visit the MathWorks Web site at `http://www.mathworks.com/products`.

Supported Serial Port Interface Standards

Over the years, several serial port interface standards have been developed. These standards include RS-232, RS-422, and RS-485 – all of which are supported by the MATLAB serial port object. Of these, the most widely used interface standard for connecting computers to peripheral devices is RS-232.

In this guide, it is assumed you are using the RS-232 standard, which is discussed in "Overview of the Serial Port" on page 9-4. Refer to your computer and device documentation to see which interface standard you can use.

Supported Platforms

The MATLAB serial port interface is supported on Microsoft Windows, Linux, and Sun Solaris platforms.

Ex.1049 / Page 616 of 748
Intel Corporation et al

Using the Examples with Your Device

Many of the examples in this section reflect specific peripheral devices connected to a PC serial port – in particular a Tektronix TDS 210 two-channel oscilloscope connected to the COM1 port. Therefore, many of the string commands are specific to this instrument.

If your peripheral device is connected to a different serial port, or if it accepts different commands, you should modify the examples accordingly.

Ex.1049 / Page 617 of 748
Intel Corporation et al

Overview of the Serial Port

This section provides an overview of the serial port. Topics include

- What is Serial Communication?
- The Serial Port Interface Standard
- Connecting Two Devices with a Serial Cable
- Serial Port Signals and Pin Assignments
- Serial Data Format
- Finding Serial Port Information for Your Platform

For many serial port applications, you can communicate with your device without detailed knowledge of how the serial port works. If your application is straightforward, or if you are already familiar with the topics mentioned above, you might want to begin with "The Serial Port Session" on page 9-19 to see how to use your serial port device with MATLAB.

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 (numerical) format or a text format.

The Serial Port Interface Standard

The serial port interface for connecting two devices is specified by the TIA/EIA-232C standard published by the Telecommunications Industry Association.

The original serial port interface standard was given by RS-232, which stands for Recommended Standard number 232. 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

Ex.1049 / Page 618 of 748
Intel Corporation et al

- 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.

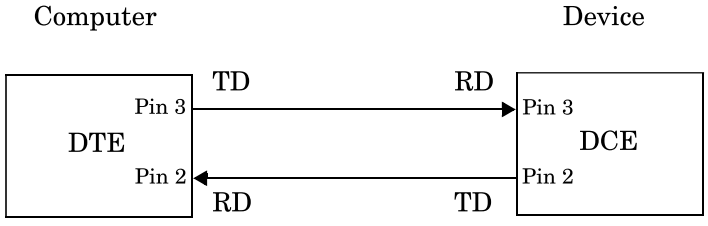
Other standards such as RS-485 define additional functionality such as higher bit transfer rates, longer cable lengths, and connections to as many as 256 devices.

Connecting Two Devices with a Serial Cable

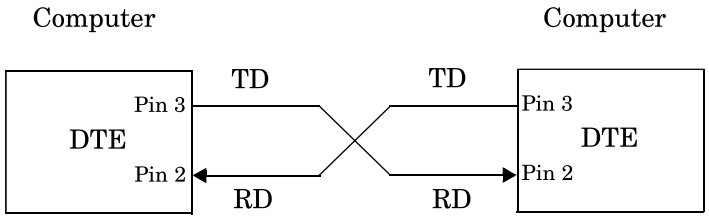
The RS-232 standard defines the two devices connected with a serial cable as the Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE). This terminology reflects the RS-232 origin as a standard for communication between a computer terminal and a modem.

Throughout this guide, your computer is considered a DTE, while peripheral devices such as modems and printers are considered DCE's. Note that many scientific instruments function as DTE's.

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 9-6 for more information about serial port pins.



If you connect two DTE's or two DCE's using a straight serial cable, then the TD pin on each device are connected to each other, and the RD pin on each device are connected to each 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.



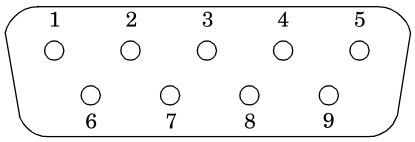
Note You can connect multiple RS-422 or RS-485 devices to a serial port. If you have an RS-232/RS-485 adaptor, then you can use the MATLAB serial port object with these devices.

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 PC's 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.

Ex.1049 / Page 620 of 748
Intel Corporation et al

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 standard for a description of the signals and pin assignments used for a 25-pin connector.

Table 9-1: 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.”

Ex.1049 / Page 621 of 748
Intel Corporation et al

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

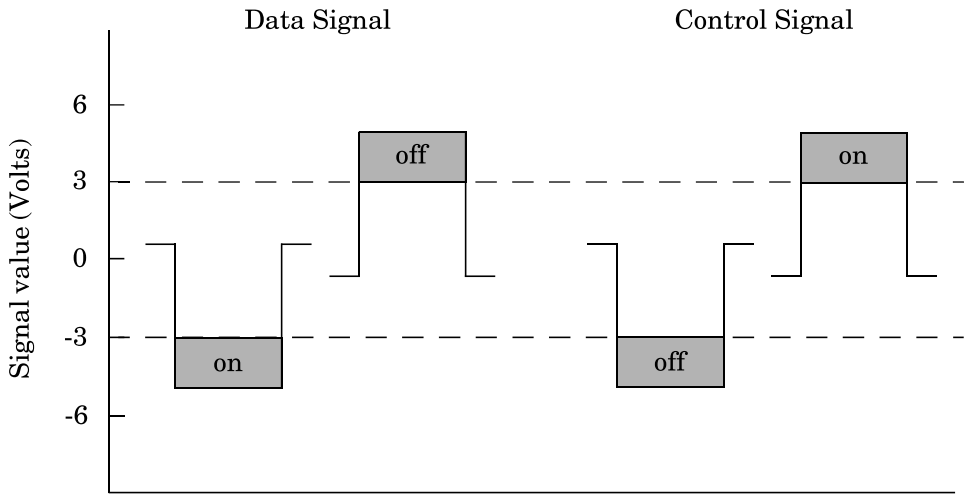
Signals can be in either an *active* state or an *inactive* state. An active state corresponds to the binary value 1, while an inactive state corresponds to the binary value 0. An active signal state is often described as *logic 1*, *on*, *true*, or a *mark*. An inactive signal state is often described 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.

Ex.1049 / Page 622 of 748
Intel Corporation et al

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

9-pin serial ports provide several control pins that:

- Signal the presence of connected devices
- Control the flow of data

The control pins include RTS and CTS, DTR and DSR, CD, and RI.

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 in MATLAB, refer to “Controlling the Flow of Data: Handshaking” on page 9-59.

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 it’s connected.
3. DCE unasserts the DSR pin when it’s 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

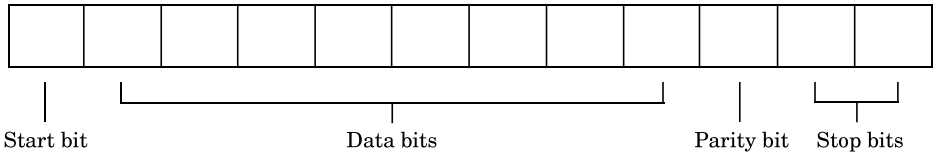
Ex.1049 / Page 624 of 748
Intel Corporation et al

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.



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 comprise 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.

Ex.1049 / Page 625 of 748
Intel Corporation et al

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 9-29.

Synchronous and Asynchronous Communication

The RS-232 standard supports two types of communication protocols: synchronous and asynchronous.

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

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

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

Note When used in this guide, the terms "synchronous" and "asynchronous" refer to whether read or write operations block access to the MATLAB command line. Refer to "Controlling Access to the MATLAB Command Line" on page 9-29 for more information.

How Are the Bits Transmitted?

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

Ex.1049 / Page 626 of 748
Intel Corporation et al

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 9-12, 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 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.

Table 9-2: Parity Types

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

Mark and space parity checking are seldom used because they offer minimal error detection. You might choose to not 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

Ex.1049 / Page 628 of 748
Intel Corporation et al

transmitting device to produce an even number of 1's. If odd parity is selected, then the parity bit is set to 1 by the transmitting device to produce an odd number of 1's.

Finding Serial Port Information for Your Platform

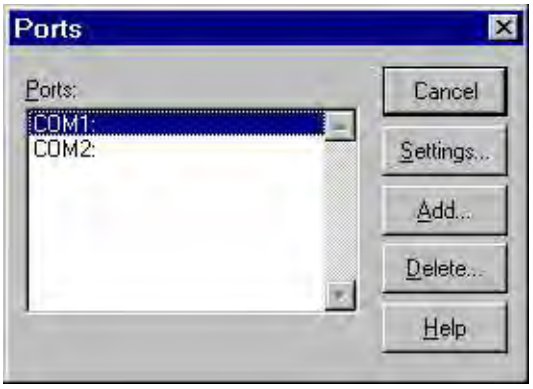
The ways to find serial port information for Windows and UNIX platforms are described below.

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.

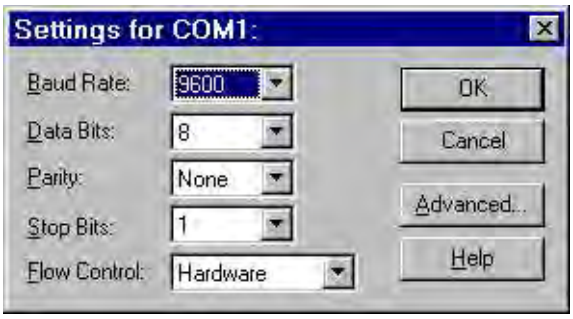
Windows Platform

You can easily access serial port information through the Windows Control Panel. You can invoke the Control Panel with the **Start** button (**Start** -> **Settings** -> **Control Panel**).

For Windows NT, you access the serial ports by selecting the Ports icon within the Control Panel. The resulting **Ports** dialog box is shown below.



To obtain information on the possible settings for COM1, select this port under the **Ports** list box and then select **Settings**.



You can access serial port information for the Windows 98 and Windows 2000 operating systems with the **System Properties** dialog box, which is available through the Control Panel.

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 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
```


Ex.1049 / Page 630 of 748
Intel Corporation et al

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

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

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

To configure the baud rate to 4800 bits per second

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

Selected Bibliography

1. TIA/EIA-232-F, *Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*.
2. Jan Axelson, *Serial Port Complete*, Lakeview Research, Madison, WI, 1998.
3. *Instrument Communication Handbook*, IOTech, Inc., Cleveland, OH, 1991.
4. *TDS 200-Series Two Channel Digital Oscilloscope Programmer Manual,* Tektronix, Inc., Wilsonville, OR.
5. *Courier High Speed Modems User's Manua*l, U.S. Robotics, Inc., Skokie, IL, 1994.
6. *Getting Started with Your AT Serial Hardware and Software for Windows 98/95*, National Instruments, Inc., Austin, TX, 1998.

Ex.1049 / Page 631 of 748
Intel Corporation et al

Getting Started with Serial I/O

To get you started with the MATLAB serial port interface, this section provides the following information:

- "Example: Getting Started" illustrates some basic serial port commands.
- "The Serial Port Session" describes the steps you use to perform any serial port task from beginning to end.
- "Configuring and Returning Properties" describes how you display serial port property names and property values, and how you assign values to properties.

Example: Getting Started

If you have a device connected to the serial port COM1 and configured for a baud rate of 4800, you can execute the following complete example.

```
s = serial('COM1');
set(s,'BaudRate',4800);
fopen(s);
fprintf(s,'*IDN?')
out = fscanf(s);
fclose(s)
delete(s)
clear s
```

The `*IDN?` command queries the device for identification information, which is returned to `out`. If your device does not support this command, or if it is connected to a different serial port, you should modify the above example accordingly.

Note `*IDN?` is one of the commands supported by the Standard Commands for Programmable Instruments (SCPI) language, which is used by many modern devices. Refer to your device documentation to see if it supports the SCPI language.

Ex.1049 / Page 632 of 748
Intel Corporation et al

The Serial Port Session

The serial port *session* comprises all the steps you are likely to take when communicating with a device connected to a serial port. These steps are:

1. **Create a serial port object** – You create a serial port object for a specific serial port using the `serial` creation function.

   You can also configure properties during object creation. In particular, you might want to configure properties associated with serial port communications such as the baud rate, the number of data bits, and so on.

2. **Connect to the device** – You connect the serial port object to the device using the `fopen` function.

   After the object is connected, you can alter device settings by configuring property values, read data, and write data.

3. **Configure properties** – To establish the desired serial port object behavior, you assign values to properties using the `set` function or dot notation.

   In practice, you can configure many of the properties at any time including during, or just after, object creation. Conversely, depending on your device settings and the requirements of your serial port application, you might be able to accept the default property values and skip this step.

4. **Write and read data** – You can now write data to the device using the `fprintf` or `fwrite` function, and read data from the device using the `fgetl`, `fgets`, `fread`, `fscanf`, or `readasync` function.

   The serial port object behaves according to the previously configured or default property values.

5. **Disconnect and clean up** – When you no longer need the serial port object, you should disconnect it from the device using the `fclose` function, remove it from memory using the `delete` function, and remove it from the MATLAB workspace using the `clear` command.

The serial port session is reinforced in many of the serial port documentation examples. Refer to “Example: Getting Started” on page 9-18 to see a basic example that uses the steps shown above.

Ex.1049 / Page 633 of 748
Intel Corporation et al

Configuring and Returning Properties

You establish the desired serial port object behavior by configuring property values. You can display or configure property values using the `set` function, the `get` function, or dot notation.

Displaying Property Names and Property Values

Once the serial port object is created, you can use the `set` function to display all the configurable properties to the command line. Additionally, if a property has a finite set of string values, then `set` also displays these values.

```
s = serial('COM1');
set(s)
    ByteOrder: [ {littleEndian} | bigEndian ]
    BytesAvailableFcn
    BytesAvailableFcnCount
    BytesAvailableFcnMode: [ {terminator} | byte ]
    ErrorFcn
    InputBufferSize
    Name
    OutputBufferSize
    OutputEmptyFcn
    RecordDetail: [ {compact} | verbose ]
    RecordMode: [ {overwrite} | append | index ]
    RecordName
    Tag
    Timeout
    TimerFcn
    TimerPeriod
    UserData

    SERIAL specific properties:
    BaudRate
    BreakInterruptFcn
    DataBits
    DataTerminalReady: [ {on} | off ]
    FlowControl: [ {none} | hardware | software ]
    Parity: [ {none} | odd | even | mark | space ]
    PinStatusFcn
    Port
    ReadAsyncMode: [ {continuous} | manual ]
```

```
    RequestToSend: [ {on} | off ]
    StopBits
    Terminator
```

You can use the `get` function to display one or more properties and their current values to the command line. To display all properties and their current values

```
get(s)
    ByteOrder = littleEndian
    BytesAvailable = 0
    BytesAvailableFcn =
    BytesAvailableFcnCount = 48
    BytesAvailableFcnMode = terminator
    BytesToOutput = 0
    ErrorFcn =
    InputBufferSize = 512
    Name = Serial-COM1
    OutputBufferSize = 512
    OutputEmptyFcn =
    RecordDetail = compact
    RecordMode = overwrite
    RecordName = record.txt
    RecordStatus = off
    Status = closed
    Tag =
    Timeout = 10
    TimerFcn =
    TimerPeriod = 1
    TransferStatus = idle
    Type = serial
    UserData = []
    ValuesReceived = 0
    ValuesSent = 0

    SERIAL specific properties:
    BaudRate = 9600
    BreakInterruptFcn =
    DataBits = 8
    DataTerminalReady = on
    FlowControl = none
```

```
    Parity = none
    PinStatus = [1x1 struct]
    PinStatusFcn =
    Port = COM1
    ReadAsyncMode = continuous
    RequestToSend = on
    StopBits = 1
    Terminator = LF
```

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

```
get(s,'OutputBufferSize')
ans =
   512
```

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

```
get(s,{'Parity','TransferStatus'})
ans =
    'none'    'idle'
```

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

```
s.Parity
ans =
none
```

Configuring Property Values

You can configure property values using the `set` function

```
set(s,'BaudRate',4800);
```

or the dot notation.

```
s.BaudRate = 4800;
```

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

```
set(s,'DataBits',7,'Name','Test1-serial')
```

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 serial port object exists – including during object creation. However, some properties are not configurable while the object is connected to the device or when recording information to disk. Refer to “Property Reference” on page 9-70 for information about when a property is configurable.

Specifying Property Names

Serial port 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 `BaudRate` property any of these ways.

```
set(s,'BaudRate',4800)
set(s,'baudrate',4800)
set(s,'BAUD',4800)
```

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

Default Property Values

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

Note Your operating system provides default values for all serial port settings such as the baud rate. However, these settings are overridden by your MATLAB code, and will have no effect on your serial port application.

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

```
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```

You can find the default value for any property in the property reference pages.

Ex.1049 / Page 637 of 748
Intel Corporation et al

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. Additionally, you can configure property values during object creation. For example, to create a serial port object associated with the serial port COM1

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

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

```
whos s
  Name      Size         Bytes  Class

  s         1x1            512  serial object

Grand total is 11 elements using 512 bytes
```

Once the serial port object is created, the properties listed below are automatically assigned values. These general purpose properties provide descriptive information about the serial port object based on the object type and the serial port.

Table 9-3: Descriptive General Purpose 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` with the `get` function.

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

Configuring Properties During Object Creation

You can configure serial port properties during object creation. `serial` accepts property names and property values in the same format as the `set` function. For example, you can specify property name/property value pairs.

```
s = serial('COM1','BaudRate',4800,'Parity','even');
```

If you specify an invalid property name, the object is not created. However, if you specify an invalid value for some properties (for example, `BaudRate` is set to 50), the object might be created but you will not be informed of the invalid value until you connect the object to the device with the `fopen` function.

The Serial Port Object Display

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

- Type the serial port object variable name at the command line.
- Exclude the semicolon when creating a serial port object.
- Exclude the semicolon when configuring properties using the dot notation.

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

```
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
```

Creating an Array of Serial Port Objects

In MATLAB, you can create an array from existing variables by concatenating those variables together. The same is true for serial port objects. For example, suppose you create the serial port objects `s1` and `s2`

```
s1 = serial('COM1');
s2 = serial('COM2');
```

You can now create a serial port object array consisting of `s1` and `s2` using the usual MATLAB syntax. To create the row array `x`

```
x = [s1 s2]

Instrument Object Array

   Index:   Type:        Status:      Name:
   1        serial       closed       Serial-COM1
   2        serial       closed       Serial-COM2
```

To create the column array `y`

```
y = [s1;s2];
```

Note that you cannot create a matrix of serial port objects. For example, you cannot create the matrix

```
z = [s1 s2;s1 s2];
??? Error using ==> serial/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 serial port objects to a function. For example, to configure the baud rate and parity for `s1` and `s2` using one call to `set`

```
set(x,'BaudRate',19200,'Parity','even')
```

Refer to the serial port function reference to see which functions accept a serial port object array as an input.

Connecting to the Device

Before you can use the serial port object to write or read data, you must connect it to your device via the serial port specified in the `serial` function. You connect a serial port object to the device with the `fopen` function.

```
fopen(s)
```

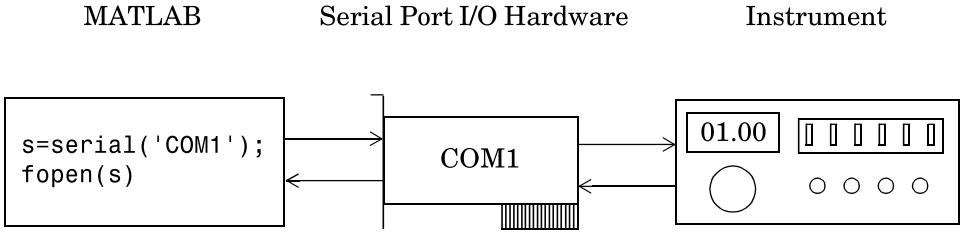
Some properties are read-only while the serial port object is connected and must be configured before using `fopen`. Examples include the `InputBufferSize` and the `OutputBufferSize` properties. Refer to “Property Reference” on page 9-70 to determine when you can configure a property.

Note You can create any number of serial port objects. However, you can connect only one serial port object to a given serial port at a time.

You can examine the `Status` property to verify that the serial port object is connected to the device.

```
s.Status
ans =
open
```

As illustrated below, the connection between the serial port object and the device is complete, and you can write and read data.



Configuring Communication Settings

Before you can write or read data, both the serial port object and the device must have identical communication settings. Configuring serial port communications involves specifying values for properties that control the baud rate and the serial data format. These properties are given below.

Table 9-4: 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

Note If the serial port object and the device communication settings are not identical, then you cannot successfully read or write data.

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

Ex.1049 / Page 642 of 748
Intel Corporation et al

Writing and Reading Data

For many serial port applications, there are three important questions that you should consider when writing or reading data:

- Will the read or write function block access to the MATLAB command line?
- Is the data to be transferred binary (numerical) or text?
- Under what conditions will the read or write operation complete?

For write operations, these questions are answered in “Writing Data” on page 9-31. For read operations, these questions are answered in “Reading Data” on page 9-36.

Example: Introduction to Writing and Reading Data

Suppose you want to return identification information for a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1. This requires writing the `*IDN?` command to the instrument using the `fprintf` function, and then reading back the result of that command using the `fscanf` function.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
out = fscanf(s)
```

The resulting identification information is shown below.

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

End the serial port session.

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

Controlling Access to the MATLAB Command Line

You control access to the MATLAB command line by specifying whether a read or write operation is *synchronous* or *asynchronous*.

A synchronous operation blocks access to the command line until the read or write function completes execution. An asynchronous operation does not block

access to the command line, and you can issue additional commands while the read or write function executes in the background.

The terms “synchronous” and “asynchronous” are often used to describe how the serial port operates at the hardware level. The RS-232 standard supports an asynchronous communication protocol. Using this protocol, each device uses its own internal clock. The data transmission is synchronized using the start bit of the bytes, while one or more stop bits indicate the end of the byte. Refer to “Serial Data Format” on page 9-11 for more information on start bits and stop bits. The RS-232 standard also supports a synchronous mode where all transmitted bits are synchronized to a common clock signal.

At the hardware level, most serial ports operate asynchronously. However, using the default behavior for many of the read and write functions, you can mimic the operation of a synchronous serial port.

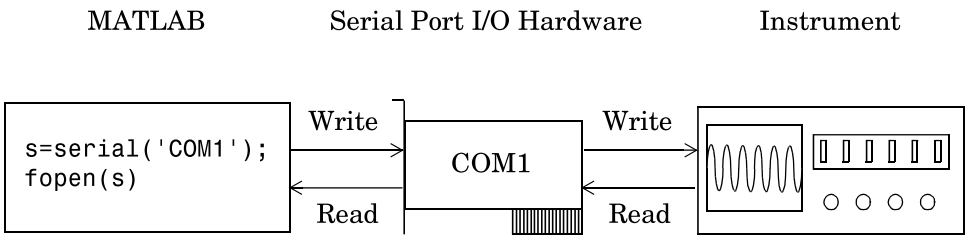
Note When used in this guide, the terms “synchronous” and “asynchronous” refer to whether read or write operations block access to the MATLAB command line. In other words, these terms describe how the software behaves, and not how the hardware behaves.

The two main advantages of writing or reading data asynchronously are:

- You can issue another command while the write or read function is executing.
- You can use all supported callback properties (see “Events and Callbacks” on page 9-48).

Ex.1049 / Page 644 of 748
Intel Corporation et al

For example, because serial ports have separate read and write pins, you can simultaneously read and write data. This is illustrated below.



Writing Data

This section describes writing data to your serial port device in three parts:

- "The Output Buffer and Data Flow" describes the flow of data from MATLAB to the device.
- "Writing Text Data" describes how to write text data (string commands) to the device.
- "Writing Binary Data" describes how to write binary (numerical) data to the device.

The functions associated with writing data are given below.

Table 9-5: Functions Associated with Writing Data

Function Name	Description
`fprintf`	Write text to the device
`fwrite`	Write binary data to the device
`stopasync`	Stop asynchronous read and write operations

Ex.1049 / Page 645 of 748
Intel Corporation et al

The properties associated with writing data are given below.

Table 9-6: 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 device

The Output Buffer and Data Flow

The output buffer is computer memory allocated by the serial port object to store data that is to be written to the device. When writing data to your device, the data flow follows these two 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 device.

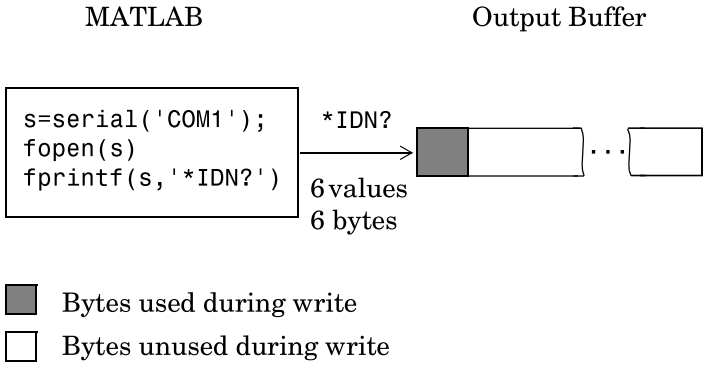
The `OutputBufferSize` property specifies the maximum number of bytes that you can store in the output buffer. The `BytesToOutput` property indicates the number of bytes currently in the output buffer. The default values for these properties are given below.

```
s = serial('COM1');
get(s,{'OutputBufferSize','BytesToOutput'})
ans =
    [512]    [0]
```

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

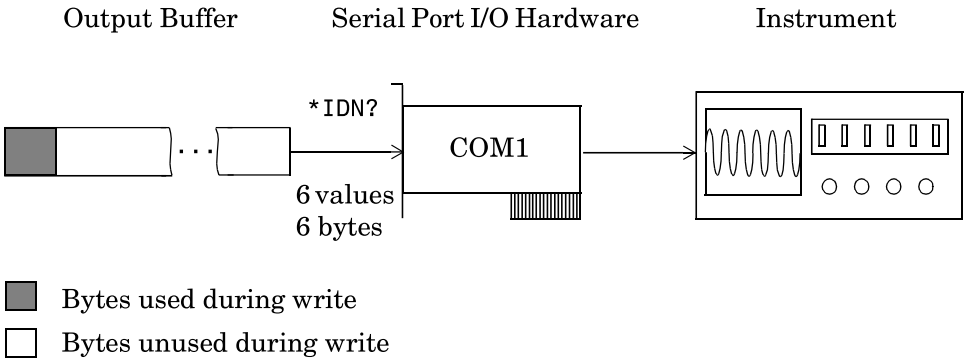
Ex.1049 / Page 646 of 748
Intel Corporation et al

For example, suppose you write the string command `*IDN?` to the TDS 210 oscilloscope using the `fprintf` function. As shown below, the string is first written to the output buffer as six values.



The `*IDN?` command consists of six values because the terminator is automatically written. Moreover, the default data format for the `fprintf` function specifies that one value corresponds to one byte. For more information about bytes and values, refer to “Bytes Versus Values” on page 9-11. `fprintf` and the terminator are discussed in “Writing Text Data” on page 9-34.

As shown below, after the string is written to the output buffer, it is then written to the device via the serial port.



Writing Text Data

You use the `fprintf` function to write text data to the device. For many devices, writing text data means writing string commands that change device settings, prepare the device to return data or status information, and so on.

For example, the `Display:Contrast` command changes the display contrast of the oscilloscope.

```
s = serial('COM1');
fopen(s)
fprintf(s,'Display:Contrast 45')
```

By default, `fprintf` writes data using the `%s\n` format because many serial port devices accept only text-based commands. However, you can specify many other formats as described in the `fprintf` reference pages.

You can verify the number of values sent to the device with the `ValuesSent` property.

```
s.ValuesSent
ans =
    20
```


Ex.1049 / Page 648 of 748
Intel Corporation et al

Note that the `ValuesSent` property value includes the terminator because each occurrence of `\n` in the command sent to the device is replaced with the `Terminator` property value.

```
s.Terminator
ans =
LF
```

The default value of `Terminator` is the line feed character. The terminator required by your device will be described in its documentation.

Synchronous Versus Asynchronous Write Operations. By default, `fprintf` operates synchronously and will block the MATLAB command line until execution completes. To write text data asynchronously to the device, you must specify `async` as the last input argument to `fprintf`.

```
fprintf(s,'Display:Contrast 45','async')
```

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

- Execute an asynchronous read operation because serial ports have separate pins for reading and writing
- Make use of all supported callback properties

You can determine which asynchronous operations are in progress with the `TransferStatus` property. If no asynchronous operations are in progress, then `TransferStatus` is `idle`.

```
s.TransferStatus
ans =
idle
```

Rules for Completing a Write Operation with fprintf. A synchronous or asynchronous write operation using `fprintf` completes when:

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

Additionally, you can stop an asynchronous write operation with the `stopasync` function.

Writing Binary Data

You use the `fwrite` function to write binary data to the device. Writing binary data means writing numerical values. A typical application for writing binary data involves writing calibration data to an instrument such as an arbitrary waveform generator.

Note Some serial port devices accept only text-based commands. These commands might use the SCPI language or some other vendor-specific language. Therefore, you might need to use the `fprintf` function for all write operations.

By default, `fwrite` translates values using the `uchar` precision. However, you can specify many other precisions as described in the reference pages for this function.

By default, `fwrite` operates synchronously. To write binary data asynchronously to the device, you must specify `async` as the last input argument to `fwrite`. For more information about synchronous and asynchronous write operations, refer to the "Writing Text Data" on page 9-34. For a description of the rules used by `fwrite` to complete a write operation, refer to its reference pages.

Reading Data

This section describes reading data from your serial port device in three parts:

- "The Input Buffer and Data Flow" describes the flow of data from the device to MATLAB.
- "Reading Text Data" describes how to read from the device, and format the data as text.
- "Reading Binary Data" describes how to read binary (numerical) data from the device.

Ex.1049 / Page 650 of 748
Intel Corporation et al

The functions associated with reading data are given below.

Table 9-7: Functions Associated with Reading Data

Function Name	Description
`fgetl`	Read one line of text from the device and discard the terminator
`fgets`	Read one line of text from the device and include the terminator
`fread`	Read binary data from the device
`fscanf`	Read data from the device, and format as text
`readasync`	Read data asynchronously from the device
`stopasync`	Stop asynchronous read and write operations

The properties associated with reading data are given below.

Table 9-8: 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 operation is continuous or manual
`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 device

Ex.1049 / Page 651 of 748
Intel Corporation et al

The Input Buffer and Data Flow

The input buffer is computer memory allocated by the serial port object to store data that is to be read from the device. When reading data from your device, the data flow follows these two steps:

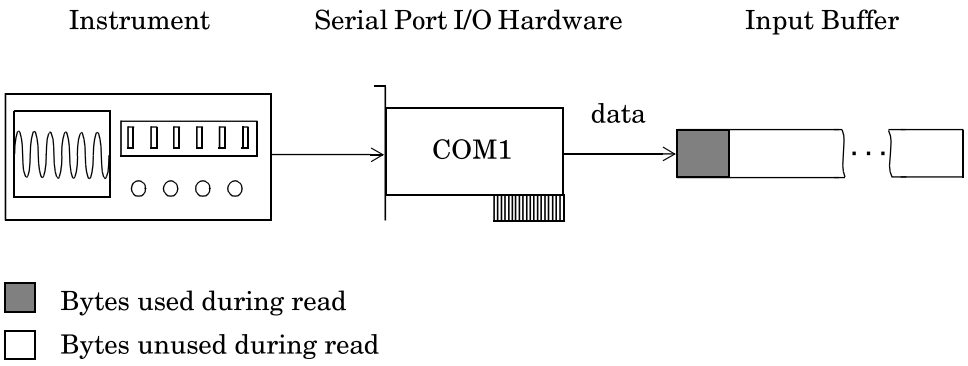
1. The data read from the device is stored in the input buffer.
2. The data in the input buffer is returned to the MATLAB variable specified by the read function.

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

```
s = serial('COM1');
get(s,{'InputBufferSize','BytesAvailable'})
ans =
    [512]    [0]
```

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

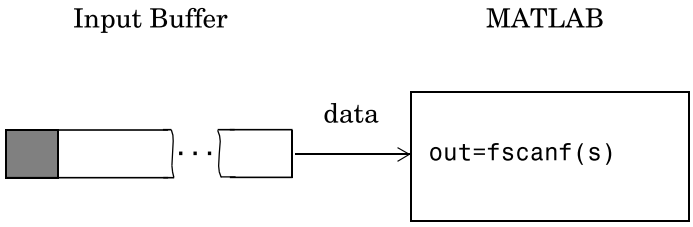
For example, suppose you use the `fscanf` function to read the text-based response of the `*IDN?` command previously written to the TDS 210 oscilloscope. As shown below, the text data is first read into to the input buffer via the serial port.



Ex.1049 / Page 652 of 748
Intel Corporation et al

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

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



Reading Text Data

You use the `fgetl`, `fgets`, and `fscanf` functions to read data from the device, and format the data as text.

For example, suppose you want to return identification information for the oscilloscope. This requires writing the `*IDN?` command to the instrument, and then reading back the result of that command.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
out = fscanf(s)
out =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

By default, `fscanf` reads data using the `%c` format because the data returned by many serial port devices is text based. However, you can specify many other formats as described in the `fscanf` reference pages.

Ex.1049 / Page 653 of 748
Intel Corporation et al

You can verify the number of values read from the device – including the terminator – with the `ValuesReceived` property.

```
s.ValuesReceived
ans =
    56
```

Synchronous Versus Asynchronous Read Operations. 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 device to determine if data is available to be read. If data is available, it is asynchronously stored in the input buffer. To transfer the data from the input buffer to MATLAB, you use one of the synchronous (blocking) read functions such as `fgetl` or `fscanf`. If data is available in the input buffer, these functions will return quickly.

```
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 device to determine if data is available to be read. To read data asynchronously, you use the `readasync` function.You then use one of the synchronous read functions to transfer data from the input buffer to MATLAB.

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

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

- Execute an asynchronous write operation because serial ports have separate pins for reading and writing
- Make use of all supported callback properties

You can determine which asynchronous operations are in progress with the `TransferStatus` property. If no asynchronous operations are in progress, then `TransferStatus` is `idle`.

```
s.TransferStatus
ans =
idle
```

Rules for Completing a Read Operation with fscanf. A read operation with `fscanf` blocks access to the MATLAB command line until:

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

Reading Binary Data

You use the `fread` function to read binary data from the device. Reading binary data means that you return numerical values to MATLAB.

For example, suppose you want to return the cursor and display settings for the oscilloscope. This requires writing the `CURSOR?` and `DISPLAY?` commands to the instrument, and then reading back the results of those commands.

```
s = serial('COM1');
fopen(s)
fprintf(s,'CURSOR?')
fprintf(s,'DISPLAY?')
```

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

Ex.1049 / Page 655 of 748
Intel Corporation et al

the device. You can verify the number of values read with the BytesAvailable property.

```
s.BytesAvailable
ans =
    69
```

You can return the data to MATLAB using any of the synchronous read functions. However, if you use `fgetl`, `fgets`, or `fscanf`, then you must issue the function twice because there are two terminators stored in the input buffer. If you use `fread`, then you can return all the data to MATLAB in one function call.

```
out = fread(s,69);
```

By default, `fread` returns numerical values in double precision arrays. However, you can specify many other precisions as described in the `fread` reference pages. You can convert the numerical data to text using the MATLAB `char` function.

```
val = char(out)'
val =
HBARS;CH1;SECONDS;-1.0E-3;1.0E-3;VOLTS;-6.56E-1;6.24E-1
YT;DOTS;0;45
```

For more information about synchronous and asynchronous read operations, refer to “Reading Text Data” on page 9-39. For a description of the rules used by `fread` to complete a read operation, refer to its reference pages.

Example: Writing and Reading Text Data

This example illustrates how to communicate with a serial port instrument by writing and reading text data.

The instrument is a Tektronix TDS 210 two-channel oscilloscope connected to the COM1 port. Therefore, many of the commands given below are specific to this instrument. A sine wave is input into channel 2 of the oscilloscope, and your job is to measure the peak-to-peak voltage of the input signal.

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

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


Ex.1049 / Page 656 of 748
Intel Corporation et al

2 **Connect to the device –** 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?')
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 MATLAB using `fscanf`.

```
ptop = fscanf(s,'%g')
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
```

Example: Parsing Input Data Using strread

This example illustrates how to use the `strread` function to parse and format data that you read from a device. `strread` is particularly useful when you want to parse a string into one or more variables, where each variable has its own specified format.

The instrument is a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1.

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

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

2 **Connect to the device –** 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 `RS232?` command to the instrument using `fprintf`, and then read back the result of the command using `fscanf`. `RS232?` 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?')
data = fscanf(s)
data =
9600;0;0;NONE;LF
```


Ex.1049 / Page 658 of 748
Intel Corporation et al

Use the `strread` function to parse and format the `data` variable into five new variables.

```
[br,sfc,hfc,par,tm] = strread(data,'%d%d%d%s%s','delimiter',';')

br =
        9600
sfc =
     0
hfc =
     0
par =
    'NONE'
tm =
    'LF'
```

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
```

Example: Reading Binary Data

This example illustrates how you can download the TDS 210 oscilloscope screen display to MATLAB. The screen display data is transferred 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.

Because the amount of data transferred is expected to be fairly large, it is asynchronously returned to the input buffer as soon as it is available from the instrument. This allows you to perform other tasks as the transfer progresses. Additionally, the scope is configured to its highest baud rate of 19,200.

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

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

2 **Configure property values –** Configure the input buffer to accept a reasonably large number of bytes, and configure the baud rate to the highest value supported by the scope.

```
s.InputBufferSize = 50000;
s.BaudRate = 19200;
```

3 **Connect to the device –** 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)
```

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

```
fprintf(s,'HARDCOPY:PORT RS232')
fprintf(s,'HARDCOPY:FORMAT BMP')
fprintf(s,'HARDCOPY START')
```

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.

```
out = fread(s,s.BytesAvailable,'uint8');
```

5 **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
```

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.

Ex.1049 / Page 660 of 748
Intel Corporation et al

4 Read the data into MATLAB using the `imread` function.

5 Scale and display the data using the `imagesc` function.

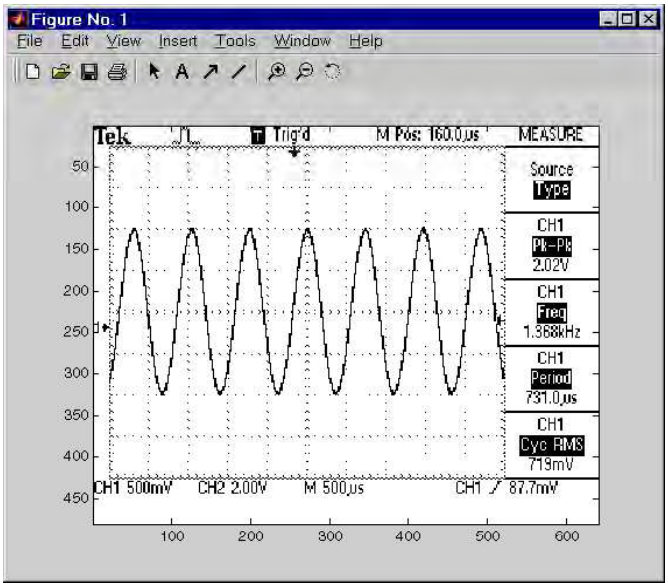
Note that the file I/O versions of the `fopen`, `fwrite`, and `fclose` functions are used.

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

Because the scope returns the screen display data using only two colors, an appropriate colormap is selected.

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

The resulting bitmap image is shown below.



Events and Callbacks

You can enhance the power and flexibility of your serial port application by using *events*. An event occurs after a condition is met and might result in one or more callbacks.

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

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

Example: Introduction to Events and Callbacks

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

```
s = serial('COM1');
fopen(s)
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @instrcallback;
fprintf(s,'*IDN?')
out = fscanf(s);
```

The resulting display from `instrcallback` is shown below.

```
BytesAvailable event occurred at 17:01:29 for the object:
Serial-COM1.
```

End the serial port session.

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

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

Ex.1049 / Page 662 of 748
Intel Corporation et al

Event Types and Callback Properties

The serial port event types and callback properties are described below.

Table 9-9: 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`

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.

Bytes-Available Event. A bytes-available event is generated immediately after a predetermined number of bytes are available in the input buffer or a terminator 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

Ex.1049 / Page 663 of 748
Intel Corporation et al