Passing Data to a Java Method

When you make a call from MATLAB to Java code, any MATLAB data types you pass in the call are converted to data types native to the Java language. MATLAB performs this conversion on each argument that is passed, except for those arguments that are already Java objects. This section describes the conversion that is performed on specific MATLAB data types and, at the end, also takes a look at how argument types affect calls made to overloaded methods.

If data is to be returned by the method being called, MATLAB receives this data and converts it to the appropriate MATLAB format wherever necessary. This process is covered in the next section entitled “Handling Data Returned from a Java Method” on page 7-56.

This section addresses the following topics:

- “Conversion of MATLAB Argument Data”
- “Passing Built-In Data Types”
- “Passing String Arguments”
- “Passing Java Objects”
- “Other Data Conversion Topics”
- “Passing Data to Overloaded Methods”

Conversion of MATLAB Argument Data

MATLAB data, passed as arguments to Java methods, are converted by MATLAB into data types that best represent the data to the Java language. The table below shows all of the MATLAB base types for passed arguments and the Java base types defined for input arguments. Each row shows a MATLAB type followed by the possible Java argument matches, from left to right in order of closeness of the match. The MATLAB types (except cell arrays) can all be

scalar (1-by-1) arrays or matrices. All of the Java types can be scalar values or arrays.

Table 7-2: Conversion of MATLAB Types to Java Types

MATLAB Argument	Closest Type (7)	Java Input Argument (Scalar or Array)					Least Close Type (1)
`double (logical)`	`boolean`	`byte`	`short`	`int`	`long`	`float`	`double`
`double`	`double`	`float`	`long`	`int`	`short`	`byte`	`boolean`
`single`	`float`	`double`					
`char`	`String`	`char`					
`uint8`	`byte`	`short`	`int`	`long`	`float`	`double`	
`uint16`	`short`	`int`	`long`	`float`	`double`		
`uint32`	`int`	`long`	`float`	`double`			
`int8`	`byte`	`short`	`int`	`long`	`float`	`double`	
`int16`	`short`	`int`	`long`	`float`	`double`		
`int32`	`int`	`long`	`float`	`double`			
`cell array of strings`	`array of String`						
`Java object`	`Object`						
`cell array of object`	`array of Object`						

Data type conversion of arguments passed to Java code will be discussed in the following three categories. MATLAB handles each category differently.

- “Passing Built-In Data Types”
- “Passing String Arguments”
- “Passing Java Objects”

Passing Built-In Data Types

Java has eight data types that are intrinsic to the language and are not represented as Java objects. These are often referred to as *built-in*, or *elemental*, data types and they include `boolean`, `byte`, `short`, `long`, `int`, `double`, `float`, and `char`. MATLAB converts its own data types to these Java built-in types according to the table, "Conversion of MATLAB Types to Java Types" on page 7-47. Built-in types are in the first 10 rows of the table.

When a Java method you are calling expects one of these data types, you can pass it the type of MATLAB argument shown in the left-most column of the table. If the method takes an array of one of these types, you can pass a MATLAB array of the data type. MATLAB converts the data type of the argument to the type assigned in the method declaration.

The MATLAB code shown below creates a top-level window frame and sets its dimensions. The call to `setBounds` passes four MATLAB scalars of the `double` type to the inherited Java Frame method, `setBounds`, that takes four arguments of the `int` type. MATLAB converts each 64-bit double data type to a 32-bit integer prior to making the call. Shown here is the `setBounds` method declaration followed by the MATLAB code that calls the method.

```
public void setBounds(int x, int y, int width, int height)

frame=java.awt.Frame;
frame.setBounds(200,200,800,400);
frame.setVisible(1);
```

Passing Built-In Types in an Array

To call a Java method with an argument defined as an *array* of a built-in type, you can create and pass a MATLAB matrix with a compatible base type. The following code defines a polygon by sending four `x` and `y` coordinates to the Polygon constructor. Two 1-by-4 MATLAB arrays of `double` are passed to `java.awt.Polygon`, which expects `integer` arrays in the first two arguments. Shown here is the Java method declaration followed by MATLAB code that calls the method, and then verifies the set coordinates.

```
public Polygon(int xpoints[], int ypoints[], int npoints)

poly = java.awt.Polygon([14 42 98 124], [55 12 -2 62], 4);
```

```
[poly.xpoints poly.ypoints]       % Verify the coordinates
ans =
14     55
42     12
98    -2
124    62
```

MATLAB Arrays Are Passed by Value

Since MATLAB arrays are passed by value, any changes that a Java method makes to them will not be visible to your MATLAB code. If you need to access changes that a Java method makes to an array, then, rather than passing a MATLAB array, you should create and pass a Java array, which is a reference. For a description of using Java arrays in MATLAB, see "Working with Java Arrays" on page 7-28.

Note Generally, it is preferable to have methods return data that has been modified using the return argument mechanism as opposed to passing a reference to that data in an argument list.

Passing String Arguments

To call a Java method that has an argument defined as an object of class `java.lang.String`, you can pass either a `String` object that was returned from an earlier Java call or a MATLAB 1-by-n character array. If you pass the character array, MATLAB converts the array to a Java object of `java.lang.String` for you.

For a programming example, see "Example – Reading a URL" on page 7-63. This shows a MATLAB character array that holds a URL being passed to the Java `URL` class constructor. The constructor, shown below, expects a Java String argument.

```
public URL(String spec) throws MalformedURLException
```

In the MATLAB call to this constructor, a character array specifying the URL is passed. MATLAB converts this array to a Java String object prior to calling the constructor.

```
url = java.net.URL(...
     'http://archive.ncsa.uiuc.edu/demoweb/')
```

Passing Strings in an Array

When the method you are calling expects an argument of an array of type `String`, you can create such an array by packaging the strings together in a MATLAB cell array. The strings can be of varying lengths since you are storing them in different cells of the array. As part of the method call, MATLAB converts the cell array to a Java array of `String` objects.

In the following example, the `echoPrompts` method of a user-written class accepts a string array argument that MATLAB converted from its original format as a cell array of strings. The parameter list in the Java method appears as follows.

```
public String[] echoPrompts(String s[])
```

You create the input argument by storing both strings in a MATLAB cell array. MATLAB converts this structure to a Java array of `String`.

```
myaccount.echoPrompts({'Username: ','Password: '})
ans =
'Username: '
'Password: '
```

Passing Java Objects

When calling a method that has an argument belonging to a particular Java class, you must pass an object that is an instance of that class. In the example below, the `add` method belonging to the `java.awt.Menu` class requires, as an argument, an object of the `java.awt.MenuItem` class. The method declaration for this is

```
public MenuItem add(MenuItem mi)
```

The example operates on the frame created in the previous example in “Passing Built-In Data Types” on page 7-48. The second, third, and fourth lines of code shown here add items to a menu to be attached to the existing window frame.

Ex.1049 / Page 526 of 748
Intel Corporation et al

In each of these calls to `menu1.add`, an object that is an instance of the `java.awt.MenuItem` Java class is passed.

```
menu1 = java.awt.Menu('File Options');
menu1.add(java.awt.MenuItem('New'));
menu1.add(java.awt.MenuItem('Open'));
menu1.add(java.awt.MenuItem('Save'));

menuBar=java.awt.MenuBar;
menuBar.add(menu1);
frame.setMenuBar(menuBar);
```

Handling Objects of Class java.lang.Object

A special case exists when the method being called takes an argument of the `java.lang.Object` class. Since this class is the root of the Java class hierarchy, you can pass objects of any class in the argument. The hash table example shown that follows, passes objects belonging to different classes to a common method, `put`, which expects an argument of `java.lang.Object`. The method declaration for `put` is

```
public synchronized Object put(Object key, Object value)
```

The following MATLAB code passes objects of different types (`boolean`, `float`, and `string`) to the `put` method.

```
hTable = java.util.Hashtable;
hTable.put(0, java.lang.Boolean('TRUE'));
hTable.put(1, java.lang.Float(41.287));
hTable.put(2, java.lang.String('test string'));

hTable                  % Verify hash table contents
hTable =
{1.0=41.287, 2.0=test string, 0.0=true}
```

When passing arguments to a method that takes `java.lang.Object`, it is not necessary to specify the class name for objects of a built-in data type. Line three, in the example above, specifies that `41.287` is an instance of class `java.lang.Float`. You can omit this and simply say, `41.287`, as shown in the following example. Thus, MATLAB will create each object for you, choosing the closest matching Java object representation for each argument.

The three calls to put from the preceding example can be rewritten as

```
hTable.put(0, 1);
hTable.put(1, 41.287);
hTable.put(2, 'test string');
```

Passing Objects in an Array

The only types of Java object arrays that you can pass to Java methods are Java arrays and MATLAB cell arrays.

If the objects have already been placed into an array, either an array returned from a Java constructor or constructed in MATLAB by the javaArray function, then you simply pass it as is in the argument to the method being called. No conversion is done by MATLAB, as this is already a Java array.

If you have objects that are not already in a Java array, then MATLAB allows you to simply pass them in a MATLAB cell array. In this case, MATLAB converts the cell array to a Java array prior to passing the argument.

The following example shows the mapPoints method of a user-written class accepting an array of class java.awt.Point. The method declaration for this is

```
public Object mapPoints(java.awt.Point p[])
```

The MATLAB code shown below creates a 2-by-2 cell array containing four Java Point objects. When the cell array is passed to the mapPoints method, MATLAB converts it to a Java array of type java.awt.Point.

```
pointObj1 = java.awt.Point(25,143);
pointObj2 = java.awt.Point(31,147);
pointObj3 = java.awt.Point(49,151);
pointObj4 = java.awt.Point(52,176);

cellArray={pointObj1, pointObj2; pointObj3, pointObj4}
cellArray =
    [1x1 java.awt.Point]    [1x1 java.awt.Point]
    [1x1 java.awt.Point]    [1x1 java.awt.Point]

testData.mapPoints(cellArray);
```

Handling a Cell Array of Java Objects

You create a cell array of Java objects by using the MATLAB syntax `{a1,a2,...}`. You index into a cell array of Java objects in the usual way, with the syntax `a{m,n,...}`.

The following example creates a cell array of two `Frame` objects, `frame1` and `frame2`, and assigns it to variable `frames`.

```
frame1 = java.awt.Frame('Frame A');
frame2 = java.awt.Frame('Frame B');

frameArray = {frame1, frame2}
frameArray =
[1x1 java.awt.Frame]    [1x1 java.awt.Frame]
```

The next statement assigns element `{1,2}` of the cell array `frameArray` to variable `f`.

```
f = frameArray {1,2}
f =
java.awt.Frame[frame2,0,0,0x0,invalid,hidden,layout =
java.awt.BorderLayout,resizable,title=Frame B]
```

Other Data Conversion Topics

There are several remaining items of interest regarding the way MATLAB converts its data to a compatible Java type. This includes how MATLAB matches array dimensions, and how it handles empty matrices and empty strings.

How Array Dimensions Affect Conversion

The term *dimension*, as used in this section, refers more to the number of subscripts required to address the elements of an array than to its length, width, and height characteristics. For example, a 5-by-1 array is referred to as having one dimension, as its individual elements can be indexed into using only one array subscript.

In converting MATLAB to Java arrays, MATLAB handles dimension in a special manner. For a MATLAB array, dimension can be considered as the number of nonsingleton dimensions in the array. For example, a 10-by-1 array has dimension 1, and a 1-by-1 array has dimension 0. In Java, dimension is

determined solely by the number of nested arrays. For example, `double[][]` has dimension 2, and `double` has dimension 0.

If the Java array's number of dimensions exactly matches the MATLAB array's number of dimensions `n`, then the conversion results in a Java array with `n` dimensions. If the Java array has fewer than `n` dimensions, the conversion drops singleton dimensions, starting with the first one, until the number of remaining dimensions matches the number of dimensions in the Java array.

Empty Matrices and Nulls

The empty matrix is compatible with any method argument for which `NULL` is a legal value in Java. The empty string (`''`) in MATLAB translates into an empty (not `NULL`) `String` object in Java.

Passing Data to Overloaded Methods

When you invoke an overloaded method on a Java object, MATLAB determines which method to invoke by comparing the arguments your call passes to the arguments defined for the methods. Note that in this discussion, the term *method* includes constructors. When it determines the method to call, MATLAB converts the calling arguments to Java method types according to Java conversion rules, except for conversions involving objects or cell arrays. See "Passing Objects in an Array" on page 7-52.

How MATLAB Determines the Method to Call

When your MATLAB function calls a Java method, MATLAB:

1. Checks to make sure that the object (or class, for a static method) has a method by that name
2. Determines whether the invocation passes the same number of arguments of at least one method with that name
3. Makes sure that each passed argument can be converted to the Java type defined for the method

If all of the preceding conditions are satisfied, MATLAB calls the method.

In a call to an overloaded method, if there is more than one candidate, MATLAB selects the one with arguments that best fit the calling arguments.

First, MATLAB rejects all methods that have any argument types that are incompatible with the passed arguments (for example, if the method has a `double` argument and the passed argument is a `char`).

Among the remaining methods, MATLAB selects the one with the highest fitness value, which is the sum of the fitness values of all its arguments. The fitness value for each argument is the fitness of the base type minus the difference between the MATLAB array dimension and the Java array dimension. (Array dimensionality is explained in "How Array Dimensions Affect Conversion" on page 7-53.) If two methods have the same fitness, the first one defined in the Java class is chosen.

Example - Calling an Overloaded Method

Suppose a function constructs a `java.io.OutputStreamWriter` object, `osw`, and then invokes a method on the object.

```
osw.write('Test data', 0, 9);
```

MATLAB finds that the class `java.io.OutputStreamWriter` defines three `write` methods.

```
public void write(int c);
public void write(char[] cbuf, int off, int len);
public void write(String str, int off, int len);
```

MATLAB rejects the first `write` method, because it takes only one argument. Then, MATLAB assesses the fitness of the remaining two `write` methods. These differ only in their first argument, as explained below.

In the first of these two `write` methods, the first argument is defined with base type, `char`. The table, Conversion of MATLAB Types to Java Types, shows that for the type of the calling argument (MATLAB `char`), Java type, `char`, has a value of 6. There is no difference between the dimension of the calling argument and the Java argument. So the fitness value for the first argument is 6.

In the other `write` method, the first argument has Java type `String`, which has a fitness value of 7. The dimension of the Java argument is 0, so the difference between it and the calling argument dimension is 1. Therefore, the fitness value for the first argument is 6.

Because the fitness value of those two `write` methods is equal, MATLAB calls the one listed first in the class definition, with `char[]` first argument.

Handling Data Returned from a Java Method

In many cases, data returned from Java is incompatible with the data types operated on within MATLAB. When this is the case, MATLAB converts the returned value to a data type native to the MATLAB language. This section describes the conversion performed on the various data types that can be returned from a call to a Java method.

This section addresses the following topics:

- "Conversion of Java Return Data"
- "Built-In Data Types"
- "Java Objects"
- "Converting Objects to MATLAB Data Types"

Conversion of Java Return Data

The following table lists Java return types and the resulting MATLAB types. For some Java base return types, MATLAB treats scalar and array returns differently, as described following the table.

Table 7-3: Conversion of Java Types to MATLAB Types

Java Return Type	If Scalar Return, Resulting MATLAB Type	If Array Return, Resulting MATLAB Type
`boolean`	`double (logical)`	`double (logical)`
`byte`	`double`	`int8`
`short`	`double`	`int16`
`int`	`double`	`int32`
`long`	`double`	`double`
`float`	`double`	`single`
`double`	`double`	`double`
`char`	`char`	`char`

Ex.1049 / Page 532 of 748
Intel Corporation et al

Built-In Data Types

Java *built-in* data types are described in "Passing Built-In Data Types" on page 7-48. Basically, this data type includes `boolean`, `byte`, `short`, `long`, `int`, `double`, `float`, and `char`. When the value returned from a method call is one of these Java built-in types, MATLAB converts it according to the table, "Conversion of Java Types to MATLAB Types" on page 7-56.

A single numeric or `boolean` value converts to a 1-by-1 matrix of `double`, which is convenient for use in MATLAB. An array of a numeric or `boolean` return values converts to an array of the closest base type, to minimize the required storage space. Array conversions are listed in the right-hand column of the table.

A return value of Java type `char` converts to a 1-by-1 matrix of `char`. And an array of Java `char` converts to a MATLAB array of that type.

Java Objects

When a method call returns Java objects, MATLAB leaves them in their original form. They remain as Java objects so you can continue to use them to interact with other Java methods.

The only exception to this is when the method returns data of type, `java.lang.Object`. This class is the root of the Java class hierarchy and is frequently used as a catchall for objects and arrays of various types. When the method being called returns a value of the `Object` class, MATLAB converts its value according to the table, "Conversion of Java Types to MATLAB Types" on page 7-56. That is, numeric and `boolean` objects such as `java.lang.Integer` or `java.lang.Boolean` convert to a 1-by-1 MATLAB matrix of `double`. Object arrays of these types convert to the MATLAB data types listed in the right-hand column of the table. Other object types are not converted.

Converting Objects to MATLAB Data Types

With the exception of objects of class `String` and class `Object`, MATLAB does not convert Java objects returned from method calls to a native MATLAB data type. If you want to convert Java object data to a form more readily usable in MATLAB, there are a few MATLAB functions that enable you to do this. These are described in the following sections.

Ex.1049 / Page 533 of 748
Intel Corporation et al

Converting to the MATLAB double Data Type

Using the `double` function in MATLAB, you can convert any Java object or array of objects to the MATLAB `double` data type. The action taken by the `double` function depends on the class of the object you specify:

- If the object is an instance of a numeric class (`java.lang.Number` or one of the classes that inherit from that class), then MATLAB uses a preset conversion algorithm to convert the object to a MATLAB `double`.
- If the object is not an instance of a numeric class, MATLAB checks the class definition to see if it implements a method called `toDouble`. Note that MATLAB uses `toDouble` to perform its conversion of Java objects to the MATLAB `double` data type. If such a method is implemented for this class, MATLAB executes it to perform the conversion.
- If you are using a class of your own design, you can write your own `toDouble` method to perform conversions on objects of that class to a MATLAB `double`. This enables you to specify your own means of data type conversion for objects belonging to your own classes.

Note If the class of the specified object is not `java.lang.Number`, does not inherit from that `java.lang.Number`, and it does not implement a `toDouble` method, then an attempt to convert the object using the `double` function results in an error in MATLAB.

The syntax for the `double` command is as follows, where `object` is a Java object or Java array of objects.

```
double(object);
```

Converting to the MATLAB char Data Type

With the MATLAB `char` function, you can convert `java.lang.String` objects and arrays to MATLAB data types. A single `java.lang.String` object converts to a MATLAB character array. An array of `java.lang.String` objects converts to a MATLAB cell array, with each cell holding a character array.

If the object specified in the `char` command is not an instance of the `java.lang.String` class, then MATLAB checks its class to see if it implements a method named `toChar`. If this is the case, then MATLAB executes the `toChar`

method of the class to perform the conversion. If you write your own class definitions, then you can make use of this feature by writing a `toChar` method that performs the conversion according to your own needs.

Note If the class of the specified object is not `java.lang.String` and it does not implement a `toChar` method, then an attempt to convert the object using the `char` function results in an error in MATLAB.

The syntax for the `char` command is as follows, where `object` is a Java object or Java array of objects.

```
char(object);
```

Converting to a MATLAB Structure

Java objects are similar to the MATLAB `structure` in that many of an object's characteristics are accessible via field names defined within the object. You may want to convert a Java object into a MATLAB `structure` to facilitate the handling of its data in MATLAB. Use the MATLAB `struct` function on the object to do this.

The syntax for the `struct` command is as follows, where `object` is a Java object or Java array of objects.

```
struct(object);
```

The following example converts a `java.awt.Polygon` object into a MATLAB `structure`. You can access the fields of the object directly using MATLAB `structure` operations. The last line indexes into the array, `pstruct.xpoints`, to deposit a new value into the third array element.

```
polygon = java.awt.Polygon([14 42 98 124], [55 12 -2 62], 4);

pstruct = struct(polygon)
pstruct =
    npoints: 4
    xpoints: [4x1 int32]
    ypoints: [4x1 int32]
```

```
pstruct.xpoints
ans =
    14
    42
    98
   124

pstruct.xpoints(3) = 101;
```

Converting to a MATLAB Cell Array

Use the `cell` function to convert a Java array or Java object into a MATLAB cell array. Elements of the resulting cell array will be of the MATLAB type (if any) closest to the Java array elements or Java object.

The syntax for the `cell` command is as follows, where `object` is a Java object or Java array of objects.

```
cell(object);
```

In the following example, a MATLAB cell array is created in which each cell holds an array of a different data type. The `cell` command used in the first line converts each type of object array into a cell array.

```
import java.lang.* java.awt.*;

% Create a Java array of double
dblArray = javaArray('java.lang.Double', 1, 10);
for m = 1:10
   dblArray(1, m) = Double(m * 7);
end

% Create a Java array of points
ptArray = javaArray('java.awt.Point', 3);
ptArray(1) = Point(7.1, 22);
ptArray(2) = Point(5.2, 35);
ptArray(3) = Point(3.1, 49);

% Create a Java array of strings
strArray = javaArray('java.lang.String', 2, 2);
strArray(1,1) = String('one');    strArray(1,2) = String('two');
strArray(2,1) = String('three');  strArray(2,2) = String('four');
```

```
% Convert each to cell arrays
cellArray = {cell(dblArray), cell(ptArray), cell(strArray)}
cellArray =
    {1x10 cell}    {3x1 cell}    {2x2 cell}

cellArray{1,1}       % Array of type double
ans =

   [7]   [14]   [21]   [28]   [35]   [42]   [49]   [56]   [63]   [70]

cellArray{1,2}       % Array of type Java.awt.Point

ans =

    [1x1 java.awt.Point]
    [1x1 java.awt.Point]
    [1x1 java.awt.Point]

cellArray{1,3}       % Array of type char array

ans =

    'one'      'two'
    'three'    'four'
```

Introduction to Programming Examples

The following programming examples demonstrate the MATLAB interface to Java classes and objects:

- "Example – Reading a URL" on page 7-63
- "Example – Finding an Internet Protocol Address" on page 7-66
- "Example – Communicating Through a Serial Port" on page 7-68
- "Example – Creating and Using a Phone Book" on page 7-73

Each example contains the following sections:

- Overview - Describes what the example does and how it uses the Java interface to accomplish it. Highlighted are the most important Java objects that are constructed and used in the example code.
- Description - provides a detailed description of all code in the example. For longer functions, the description is divided into functional sections that focus on a few statements.
- Running the Example - Shows a sample of the output from execution of the example code.

The example descriptions concentrate on the Java-related functions. For information on other MATLAB programming constructs, operators, and functions used in the examples, see the applicable sections in the MATLAB documentation.

Ex.1049 / Page 538 of 748
Intel Corporation et al

Example – Reading a URL

This program, `URLdemo`, opens a connection to a web site specified by a URL (Uniform Resource Locator), for the purpose of reading text from a file at that site. It constructs an object of the Java API class, `java.net.URL`, which enables convenient handling of URLs. Then, it calls a method on the URL object, to open a connection.

To read and display the lines of text at the site, `URLdemo` uses classes from the Java I/O package `java.io`. It creates an `InputStreamReader` object, and then uses that object to construct a `BufferedReader` object. Finally, it calls a method on the `BufferedReader` object to read the specified number of lines from the site.

Description of URLdemo

The major tasks performed by `URLdemo` are:

1. Construct a URL Object

The example first calls a constructor on `java.net.URL` and assigns the resulting object to variable `url`. The `URL` constructor takes a single argument, the name of the URL to be accessed, as a string. The constructor checks whether the input URL has a valid form.

```
url = java.net.URL(...
  'http://www.mathworks.com/support/tech-notes/1100/1109.shtml')
```

2. Open a Connection to the URL

The second statement of the example calls the method, `openStream`, on the `URL` object `url`, to establish a connection with the web site named by the object. The method returns an `InputStream` object to variable, `is`, for reading bytes from the site.

```
is = openStream(url)
```

3. Set Up a Buffered Stream Reader

The next two lines create a buffered stream reader for characters. The `java.io.InputStreamReader` constructor is called with the input stream `is`, to return to variable `isr` an object that can read characters. Then, the `java.io.BufferedReader` constructor is called with `isr`, to return a

`BufferedReader` object to variable `br`. A buffered reader provides for efficient reading of characters, arrays, and lines.

```
isr = java.io.InputStreamReader(is)
br = java.io.BufferedReader(isr)
```

4. Read and Display Lines of Text

The final statements read the initial lines of HTML text from the site, displaying only the first 4 lines that contain meaningful text. Within the MATLAB `for` statements, the `BufferedReader` method `readLine` reads each line of text (terminated by a `return` and/or line feed character) from the site.

```
for k = 1:108           % Skip initial HTML formatting lines
   s = readLine(br);
end

for k = 1:4             % Read the first 4 lines of text
   s = readLine(br)
end
```

Running the Example

When you run this example, you see output similar to the following.

```
url =
http://www.mathworks.com/support/tech-notes/1100/1109.shtml
is =
ice.net.CachedInputStream@7e700c
isr =
java.io.InputStreamReader@7e6696
br =
java.io.BufferedReader@7e5d1b
s =
      <p class="standard">This technical note provides an
introduction to vectorization techniques.
s =
      In order to understand some of the tricks available, an
introduction to
```

```
s =
      MATLAB indexing is provided. Then several vectorization
techniques are
s =
      discussed, in order of simplest to most complicated.
```

Example – Finding an Internet Protocol Address

The `resolveip` function returns either the name or address of an IP (internet protocol) host. If you pass `resolveip` a hostname, it returns the IP address. If you pass `resolveip` an IP address, it returns the hostname. The function uses the Java API class `java.net.InetAddress`, which enables you to find an IP address for a hostname, or the hostname for a given IP address, without making DNS calls.

`resolveip` calls a `static` method on the `InetAddress` class to obtain an `InetAddress` object. Then, it calls accessor methods on the `InetAddress` object to get the hostname and IP address for the input argument. It displays either the hostname or the IP address, depending on the program input argument.

Description of resolveip

The major tasks performed by `resolveip` are:

1. Create an InetAddress Object

Instead of constructors, the `java.net.InetAddress` class has `static` methods that return an instance of the class. The `try` statement calls one of those methods, `getByName`, passing the `input` argument that the user has passed to `resolveip`. The input argument can be either a hostname or an IP address. If `getByName` fails, the `catch` statement displays an `error` message.

```
function resolveip(input)
try
   address = java.net.InetAddress.getByName(input);
catch
   error(sprintf('Unknown host %s.', input));
end
```

2. Retrieve the Hostname and IP Address

The example uses calls to the `getHostName` and `getHostAddress` accessor functions on the `java.net.InetAddress` object, to obtain the hostname and IP address, respectively. These two functions return objects of class `java.lang.String`, so we use the `char` function to convert them to character arrays.

```
hostname = char(address.getHostName);
ipaddress = char(address.getHostAddress);
```

3. Display the Hostname or IP Address

The example uses the MATLAB `strcmp` function to compare the `input` argument to the resolved IP address. If it matches, MATLAB displays the hostname for the internet address. If the input does not match, MATLAB displays the IP address.

```
if strcmp(input,ipaddress)
   disp(sprintf('Host name of %s is %s', input, hostname));
else
   disp(sprintf('IP address of %s is %s', input, ipaddress));
end;
```

Running the Example

Here is an example of calling the `resolveip` function with a hostname.

```
resolveip ('www.mathworks.com')
IP address of www.mathworks.com is 144.212.100.10
```

Here is a call to the function with an IP address.

```
resolveip ('144.212.100.10')
Host name of 144.212.100.10 is www.mathworks.com
```


Ex.1049 / Page 543 of 748
Intel Corporation et al

Example – Communicating Through a Serial Port

The `serialexample` program uses classes of the Java API `javax.comm` package, which support access to communications ports. After defining port configuration variables, `serialexample` constructs a `javax.comm.CommPortIdentifier` object, to manage the serial communications port. The program calls the `open` method on that object to return an object of the `javax.comm.SerialPort` class, which describes the low-level interface to the `COM1` serial port, assumed to be connected to a Tektronix oscilloscope. (The example can be run without an oscilloscope.) The `serialexample` program then calls several methods on the `SerialPort` object to configure the serial port.

The `serialexample` program uses the I/O package `java.io`, to write to and read from the serial port. It calls a `static` method to return an `OutputStream` object for the serial port. It then passes that object to the constructor for `java.io.OutputStreamWriter`. It calls the `write` method on the `OutputStreamWriter` object to write a command to the serial port, which sets the contrast on the oscilloscope. It calls `write` again to write a command that checks the contrast. It then constructs an object of the `java.io.InputStreamWriter` class to read from the serial port.

It calls another `static` method on the SerialPort object to return an `OutputStream` object for the serial port. It calls a method on that object to get the number of bytes to read from the port. It passes the InputStream object to the constructor for `java.io.OutputStreamWriter`. Then, if there is data to read, it calls the `read` method on the `OutputStreamWriter` object to read the contrast data returned by the oscilloscope.

Note MATLAB also provides built-in serial port support, described in Chapter 9, “Serial Port I/O”.

Description of Serial Example

The major tasks performed by `serialexample` are:

1. Define Variables for Serial Port Configuration and Output

The first five statements define variables for configuring the serial port. The first statement defines the baud rate to be 9600, the second defines number of data bits to be 8, and the third defines the number of stop bits to be 1. The fourth statement defines parity to be off, and the fifth statement defines flow control (handshaking) to be off.

```
SerialPort_BAUD_9600 = 9600;
SerialPort_DATABITS_8 = 8;
SerialPort_STOPBITS_1 = 1;
SerialPort_PARITY_NONE = 0;
SerialPort_FLOWCTRL_NONE = 0;
```

The last variable definition sets the terminator character for writing to the serial port, to a carriage return.

```
terminator = char(13);
```

2. Create a CommPortIdentifier Object

Instead of constructors, the `javax.comm.CommPortIdentifier` class has `static` methods that return an instance of the class. The example calls one of these, `getPortIdentifier`, to return a `CommPortIdentifier` object for port `COM1`.

```
commPort = ...
javax.comm.CommPortIdentifier.getPortIdentifier('COM1');
```

3. Open the Serial Port

The example opens the serial port, by calling `open` on the `CommPortIdentifier` object `commPort`. The `open` call returns a `SerialPort` object, assigning it to `serialPort`. The first argument to `open` is the name (owner) for the port, the second argument is the name for the port, and the third argument is the number of milliseconds to wait for the open.

```
serialPort = open(commPort, 'serial', 1000);
```

4. Configure the Serial Port

The next three statements call configuration methods on the `SerialPort` object `serialPort`. The first statement calls `setSerialPortParams` to set the baud rate, data bits, stop bits, and parity. The next two statements call `setFlowControlMode` to set the flow control, and then `enableReceiveTimeout` to set the timeout for receiving data.

```
setSerialPortParams(serialPort, SerialPort_BAUD_9600,...
SerialPort_DATABITS_8, SerialPort_STOPBITS_1,...
SerialPort_PARITY_NONE);
setFlowControlMode(serialPort, SerialPort_FLOWCTRL_NONE);
enableReceiveTimeout(serialPort, 1000);
```

5. Set Up an Output Stream Writer

The example then calls a constructor to create and open a `java.io.OutputStreamWriter` object. The constructor call passes the `java.io.OutputStream` object, returned by a call to the `getOutputStream` method `serialPort`, and assigns the `OutputStreamWriter` object to `out`.

```
out = java.io.OutputStreamWriter(getOutputStream(serialPort));
```

6. Write Data to Serial Port and Close Output Stream

The example writes a string to the serial port, by calling `write` on the object `out`. The string is formed by concatenating (with MATLAB [ ] syntax) a command to set the oscilloscope's contrast to 45, with the command terminator that is required by the instrument. The next statement calls `flush` on `out` to flush the output stream.

```
write(out, ['Display:Contrast 45' terminator]);
flush(out);
```

Then, the example again calls `write` on `out` to send another string to the serial port. This string is a query command, to determine the oscilloscope's contrast setting, concatenated with the command terminator. The example then calls `close` on the output stream.

```
write(out, ['Display:Contrast?' terminator]);
close(out);
```


Ex.1049 / Page 546 of 748
Intel Corporation et al

7. Open an Input Stream and Determine Number of Bytes to Read

To read the data expected from the oscilloscope in response to the contrast query, the example opens an input stream by calling the `static` method, `InputStream.getInputStream`, to obtain an `InputStream` object for the serial port. Then, the example calls the method `available` on the `InputStream` object, `in`, and assigns the returned number of bytes to `numAvail`.

```
in = getInputStream(serialPort);
numAvail = available(in);
```

8. Create an Input Stream Reader for the Serial Port

The example then calls a `java.io.InputStreamReader` constructor, with the `InputStream` object, `in`, and assigns the new object to `reader`.

```
reader = java.io.InputStreamReader(in);
```

9. Read Data from Serial Port and Close Reader

The example reads from the serial port, by calling the `read` method on the `InputStreamReader` object reader for each available byte. The `read` statement uses MATLAB array concatenation to add each newly read byte to the array of bytes already read. After reading the data, the example calls `close` on `reader` to close the input stream reader.

```
result = [];
for k = 1:numAvail
   result = [result read(reader)];
end
close(reader);
```

10. Close the Serial Port

The example closes the serial port, by calling `close` on the `serialPort` object.

```
close(serialPort);
```

11. Convert Input Argument to a MATLAB Character Array

The last statement of the example uses the MATLAB function, `char`, to convert the array input bytes (integers) to an array of characters:

```
result = char(result);
```

Running the serialexample Program

The value of `result` depends upon whether your system's COM1 port is cabled to an oscilloscope. If you have run the example with an oscilloscope, you see the result of reading the serial port.

```
result =
45
```

If you run the example without an oscilloscope attached, there is no data to read. In that case, you see an empty character array.

```
result =
''
```

Example – Creating and Using a Phone Book

The example's main function, `phonebook`, can be called either with no arguments, or with one argument, which is the key of an entry that exists in the phone book. The function first determines the directory to use for the phone book file.

If no phone book file exists, it creates one by constructing a `java.io.FileOutputStream` object, and then closing the output stream. Next, it creates a data dictionary by constructing an object of the Java API class, `java.util.Properties`, which is a subclass of `java.util.Hashtable` for storing key/value pairs in a hash table. For the `phonebook` program, the key is a name, and the value is one or more telephone numbers.

The `phonebook` function creates and opens an input stream for reading by constructing a `java.io.FileInputStream` object. It calls load on that object to load the hash table contents, if it exists. If the user passed the key to an entry to look up, it looks up the entry by calling `pb_lookup`, which finds and displays it. Then, the `phonebook` function returns.

If `phonebook` was called without the name argument, it then displays a textual menu of the available phone book actions:

- Look up an entry
- Add an entry
- Remove an entry
- Change the phone number(s) in an entry
- List all entries

The menu also has a selection to exit the program. The function uses MATLAB functions to display the menu and to input the user selection.

The `phonebook` function iterates accepting user selections and performing the requested phone book action until the user selects the menu entry to exit. The `phonebook` function then opens an output stream for the file by constructing a `java.io.FileOutputStream` object. It calls `save` on the object to write the current data dictionary to the phone book file. It finally closes the output stream and returns.

Description of Function phonebook

The major tasks performed by phonebook are:

1. Determine the Data Directory and Full Filename

The first statement assigns the phone book filename, 'myphonebook', to the variable pbname. If the phonebook program is running on a PC, it calls the java.lang.System static method getProperty to find the directory to use for the data dictionary. This will be set to the user's current working directory. Otherwise, it uses MATLAB function getenv to determine the directory, using the system variable HOME which you can define beforehand to anything you like. It then assigns to pbname the full pathname, consisting of the data directory and filename 'myphonebook'.

```
function phonebook(varargin)
pbname = 'myphonebook'; % name of data dictionary
if ispc
   datadir = char(java.lang.System.getProperty('user.dir'));
else
   datadir = getenv('HOME');
end;
pbname = fullfile(datadir, pbname);
```

2. If Needed, Create a File Output Stream

If the phonebook file does not already exist, phonebook asks the user whether to create a new one. If the user answers y, phonebook creates a new phone book by constructing a FileOutputStream object. In the try clause of a try-catch block, the argument pbname passed to the FileOutputStream constructor is the full name of the file that the constructor creates and opens. The next statement closes the file by calling close on the FileOutputStream object FOS. If the output stream constructor fails, the catch statement prints a message and terminates the program.

```
if ~exist(pbname)
   disp(sprintf('Data file %s does not exist.', pbname));
   r = input('Create a new phone book (y/n)?','s');
   if r == 'y',
      try
         FOS = java.io.FileOutputStream(pbname);
         FOS.close
      catch
```

```
            error(sprintf('Failed to create %s', pbname));
         end;
      else
         return;
      end;
   end;
```

3. Create a Hash Table

The example constructs a `java.util.Properties` object to serve as the hash table for the data dictionary.

```
   pb_htable = java.util.Properties;
```

4. Create a File Input Stream

In a `try` block, the example invokes a `FileInputStream` constructor with the name of the phone book file, assigning the object to `FIS`. If the call fails, the `catch` statement displays an `error` message and terminates the program.

```
   try
      FIS = java.io.FileInputStream(pbname);
   catch
      error(sprintf('Failed to open %s for reading.', pbname));
   end;
```

5. Load the Phone Book Keys and Close the File Input Stream

The example calls `load` on the `FileInputStream` object `FIS`, to load the phone book keys and their values (if any) into the hash table. It then closes the file input stream.

```
   pb_htable.load(FIS);
   FIS.close;
```

6. Display the Action Menu and Get the User's Selection

Within a `while` loop, several `disp` statements display a menu of actions that the user can perform on the phone book. Then, an `input` statement requests the user's typed selection.

```
   while 1
      disp ' '
      disp ' Phonebook Menu:'
```

```
    disp ' '
    disp ' 1. Look up a phone number'
    disp ' 2. Add an entry to the phone book'
    disp ' 3. Remove an entry from the phone book'
    disp ' 4. Change the contents of an entry in the phone book'
    disp ' 5. Display entire contents of the phone book'
    disp ' 6. Exit this program'
    disp ' '
   s = input('Please type the number for a menu selection: ','s');
```

7. Invoke the Function to Perform A Phone Book Action

Still within the `while` loop, a `switch` statement provides a case to handle each user selection. Each of the first five cases invokes the function to perform a phone book action.

Case 1 prompts for a name that is a key to an entry. It calls `isempty` to determine whether the user has entered a name. If a name has not been entered, it calls `disp` to display an error message. If a name has been input, it passes it to `pb_lookup`. The `pb_lookup` routine looks up the entry and, if it finds it, displays the entry contents.

```
switch s
   case '1',
      name = input('Enter the name to look up: ','s');
      if isempty(name)
         disp 'No name entered'
      else
         pb_lookup(pb_htable, name);
      end;
```

Case 2 calls `pb_add`, which prompts the user for a new entry and then adds it to the phone book.

```
case '2',
   pb_add(pb_htable);
```

Case 3 uses `input` to prompt for the name of an entry to remove. If a name has not been entered, it calls `disp` to display an error message. If a name has been input, it passes it to `pb_remove`.

```
case '3',
   name=input('Enter the name of the entry to remove: ', 's');
   if isempty(name)
      disp 'No name entered'
   else
      pb_remove(pb_htable, name);
   end;
```

Case 4 uses `input` to prompt for the name of an entry to change. If a name has not been entered, it calls `disp` to display an error message. If a name has been input, it passes it to `pb_change`.

```
case '4',
   name=input('Enter the name of the entry to change: ', 's');
   if isempty(name)
      disp 'No name entered'
   else
      pb_change(pb_htable, name);
   end;
```

Case 5 calls `pb_listall` to display all entries.

```
case '5',
   pb_listall(pb_htable);
```

8. Exit by Creating an Output Stream and Saving the Phone Book

If the user has selected case `6` to exit the program, a `try` statement calls the constructor for a `FileOuputStream` object, passing it the name of the phone book. If the constructor fails, the `catch` statement displays an error message.

If the object is created, the next statement saves the phone book data by calling `save` on the `Properties` object `pb_htable`, passing the `FileOutputStream`

object FOS and a descriptive header string. It then calls close on the FileOutputStream object, and returns.

```
      case '6',
         try
            FOS = java.io.FileOutputStream(pbname);
         catch
            error(sprintf('Failed to open %s for writing.',...
                          pbname));
         end;
         pb_htable.save(FOS,'Data file for phonebook program');
         FOS.close;
         return;
      otherwise
         disp 'That selection is not on the menu.'
      end;
   end;
```

Description of Function pb_lookup

Arguments passed to pb_lookup are the Properties object pb_htable and the name key for the requested entry. The pb_lookup function first calls get on pb_htable with the name key, on which support function pb_keyfilter is called to change spaces to underscores. The get method returns the entry (or null, if the entry is not found) to variable entry. Note that get takes an argument of type java.lang.Object and also returns an argument of that type. In this invocation, the key passed to get and the entry returned from it are actually character arrays.

pb_lookup then calls isempty to determine whether entry is null. If it is, it uses disp to display an message stating that the name was not found. If entry is not null, it calls pb_display to display the entry.

```
function pb_lookup(pb_htable,name)
entry = pb_htable.get(pb_keyfilter(name));
if isempty(entry),
   disp(sprintf('The name %s is not in the phone book',name));
else
   pb_display(entry);
end
```

Description of Function pb_add

1. Input the Entry to Add

The `pb_add` function takes one argument, the `Properties` object `pb_htable`. `pb_add` uses `disp` to prompt for an entry. Using the up-arrow (`^`) character as a line delimiter, `input` inputs a name to the variable `entry`. Then, within a while loop, it uses input to get another line of the entry into variable `line`. If the line is empty, indicating that the user has finished the entry, the code breaks out of the while loop. If the line is not empty, the else statement appends line to entry and then appends the line delimiter. At the end, the `strcmp` checks the possibility that no input was entered and, if that is the case, returns.

```
function pb_add(pb_htable)
disp 'Type the name for the new entry, followed by Enter.'
disp 'Then, type the phone number(s), one per line.'
disp 'To complete the entry, type an extra Enter.'
name = input(':: ','s');
entry=[name '^'];
while 1
   line = input(':: ','s');
   if isempty(line)
      break;
   else
      entry=[entry line '^'];
   end;
end;

if strcmp(entry, '^')
   disp 'No name entered'
   return;
end;
```

2. Add the Entry to the Phone Book

After the input has completed, `pb_add` calls `put` on `pb_htable` with the hash key `name` (on which `pb_keyfilter` is called to change spaces to underscores) and `entry`. It then displays a message that the entry has been added.

```
pb_htable.put(pb_keyfilter(name),entry);
disp ' '
disp(sprintf('%s has been added to the phone book.', name));
```

Description of Function pb_remove

1. Look For the Key in the Phone Book

Arguments passed to `pb_remove` are the `Properties` object `pb_htable` and the `name` key for the entry to remove. The `pb_remove` function calls `containsKey` on `pb_htable` with the `name` key, on which support function `pb_keyfilter` is called to change spaces to underscores. If name is not in the phone book, `disp` displays a message and the function returns.

```
function pb_remove(pb_htable,name)
if ~pb_htable.containsKey(pb_keyfilter(name))
   disp(sprintf('The name %s is not in the phone book',name))
   return
end;
```

2. Ask for Confirmation and If Given, Remove the Key

If the key is in the hash table, `pb_remove` asks for user confirmation. If the user confirms the removal by entering `y`, `pb_remove` calls `remove` on `pb_htable` with the (filtered) `name` key, and displays a message that the entry has been removed. If the user enters `n`, the removal is not performed and `disp` displays a message that the removal has not been performed.

```
r = input(sprintf('Remove entry %s (y/n)? ',name), 's');
if r == 'y'
   pb_htable.remove(pb_keyfilter(name));
   disp(sprintf('%s has been removed from the phone book',name))
else
   disp(sprintf('%s has not been removed',name))
end;
```

Description of Function pb_change

1. Find the Entry to Change, and Confirm

Arguments passed to `pb_change` are the `Properties` object `pb_htable` and the `name` key for the requested entry. The `pb_change` function calls `get` on `pb_htable` with the `name` key, on which `pb_keyfilter` is called to change spaces to underscores. The `get` method returns the entry (or `null`, if the entry is not found) to variable `entry`. `pb_change` calls `isempty` to determine whether the entry is empty. If the entry is empty, `pb_change` displays a message that

the name will be added to the phone book, and allows the user to enter the phone number(s) for the entry.

If the entry is found, in the `else` clause, `pb_change` calls `pb_display` to display the entry. It then uses input to ask the user to confirm the replacement. If the user enters anything other than `y`, the function returns.

```
function pb_change(pb_htable,name)
entry = pb_htable.get(pb_keyfilter(name));
if isempty(entry)
   disp(sprintf('The name %s is not in the phone book', name));
   return;
else
   pb_display(entry);
   r = input('Replace phone numbers in this entry (y/n)? ','s');
   if r ~= 'y'
      return;
   end;
end;
```

2. Input New Phone Number(s) and Change the Phone Book Entry

`pb_change` uses `disp` to display a prompt for new phone number(s). Then, `pb_change` inputs data into variable `entry`, with the same statements described in “1. Input the Entry to Add” on page 7-79.

Then, to replace the existing entry with the new one, `pb_change` calls `put` on `pb_htable` with the (filtered) key `name` and the new entry. It then displays a message that the entry has been changed.

```
disp 'Type in the new phone number(s), one per line.'
disp 'To complete the entry, type an extra Enter.'
disp(sprintf(':: %s', name));
entry=[name '^'];
while 1
   line = input(':: ','s');
   if isempty(line)
      break;
   else
      entry=[entry line '^'];
   end;
end;
```

```
pb_htable.put(pb_keyfilter(name),entry);
disp ' '
disp(sprintf('The entry for %s has been changed', name));
```

Description of Function pb_listall

The `pb_listall` function takes one argument, the `Properties` object `pb_htable`. The function calls `propertyNames` on the `pb_htable` object to return to `enum` a `java.util.Enumeration` object, which supports convenient enumeration of all the keys. In a `while` loop, `pb_listall` calls `hasMoreElements` on `enum`, and if it returns true, `pb_listall` calls `nextElement` on `enum` to return the next key. It then calls `pb_display` to display the key and entry, which it retrieves by calling `get` on `pb_htable` with the key.

```
function pb_listall(pb_htable)
enum = pb_htable.propertyNames;
while enum.hasMoreElements
   key = enum.nextElement;
   pb_display(pb_htable.get(key));
end;
```

Description of Function pb_display

The `pb_display` function takes an argument `entry`, which is a phone book entry. After displaying a horizontal line, `pb_display` calls MATLAB function `strtok` to extract the first line the entry, up to the line delimiter (`^`), into `t` and and the remainder into `r`. Then, within a `while` loop that terminates when `t` is empty, it displays the current line in `t`. Then it calls strtok to extract the next line from `r`, into `t`. When all lines have been displayed, `pb_display` indicates the end of the entry by displaying another horizontal line.

```
function pb_display(entry)
disp ' '
disp '-------------------------'
[t,r] = strtok(entry,'^');
while ~isempty(t)
   disp(sprintf(' %s',t));
   [t,r] = strtok(r,'^');
end;
disp '-------------------------'
```

Description of Function pb_keyfilter

The `pb_keyfilter` function takes an argument `key`, which is a name used as a key in the hash table, and either filters it for storage or unfilters it for display. The filter, which replaces each space in the key with an underscore (_), makes the key usable with the methods of `java.util.Properties`.

```
function out = pb_keyfilter(key)
if ~isempty(findstr(key,' '))
   out = strrep(key,' ','_');
else
   out = strrep(key,'_',' ');
end;
```

Running the phonebook Program

In this sample run, a user invokes `phonebook` with no arguments. The user selects menu action `5`, which displays the two entries currently in the phone book (all entries are fictitious). Then, the user selects `2`, to add an entry. After adding the entry, the user again selects `5`, which displays the new entry along with the other two entries.

```
Phonebook Menu:

     1. Look up a phone number
     2. Add an entry to the phone book
     3. Remove an entry from the phone book
     4. Change the contents of an entry in the phone book
     5. Display entire contents of the phone book
     6. Exit this program

Please type the number for a menu selection: 5

-------------------------
 Sylvia Woodland
 (508) 111-3456
-------------------------

-------------------------
 Russell Reddy
 (617) 999-8765
-------------------------
```

```
Phonebook Menu:

     1. Look up a phone number
     2. Add an entry to the phone book
     3. Remove an entry from the phone book
     4. Change the contents of an entry in the phone book
     5. Display entire contents of the phone book
     6. Exit this program

Please type the number for a menu selection: 2

Type the name for the new entry, followed by Enter.
Then, type the phone number(s), one per line.
To complete the entry, type an extra Enter.
:: BriteLites Books
:: (781) 777-6868
::

BriteLites Books has been added to the phone book.

Phonebook Menu:

     1. Look up a phone number
     2. Add an entry to the phone book
     3. Remove an entry from the phone book
     4. Change the contents of an entry in the phone book
     5. Display entire contents of the phone book
     6. Exit this program

Please type the number for a menu selection: 5

-------------------------
 BriteLites Books
 (781) 777-6868
-------------------------


-------------------------
 Sylvia Woodland
```

```
 (508) 111-3456
-------------------------

-------------------------
 Russell Reddy
 (617) 999-8765
-------------------------
```

Ex.1049 / Page 562 of 748
Intel Corporation et al

8

COM and DDE Support

Component Object Model, or COM, is a set of object-oriented technologies and tools that allow software developers to integrate application-specific components from different vendors into their own application solution. COM support in MATLAB enables you to interact with contained controls or server processes, or to configure MATLAB as a computational server controlled by your client application programs.

Dynamic Data Exchange, or DDE, is a feature of Microsoft Windows that allows two programs to share data or send commands directly to each other. MATLAB provides functions that use this data exchange to enable access between MATLAB and other Windows applications in a wide range of contexts.

Introducing MATLAB COM Integration (p. 8-2)	COM Concepts, and an overview of COM support in MATLAB
MATLAB COM Client Support (p. 8-8)	How to create COM objects and use properties, methods, and events
Additional COM Client Information (p. 8-32)	COM collections, converting data from MATLAB to COM, and using MATLAB as a DCOM server client
MATLAB Automation Server Support (p. 8-36)	Using MATLAB as a COM automation server
Additional Automation Server Information (p. 8-40)	Starting the MATLAB server; shared and dedicated servers; using MATLAB as a DCOM server
Dynamic Data Exchange (DDE) (p. 8-42)	DDE concepts; using MATLAB as a client or a server; DDE advisory links

Introducing MATLAB COM Integration

The *Component Object Model*, or COM, provides a framework for integrating reusable, binary software components into an application. Because components are implemented with compiled code, the source code may be written in any of the many programming languages that support COM. Upgrades to applications are simplified, as components can be simply swapped without the need to recompile the entire application. In addition, a component's location is transparent to the application, so components may be relocated to a separate process or even a remote system without having to modify the application.

Using COM, developers and end users can select application-specific components produced by different vendors and integrate them into a complete application solution. For example, a single application may require database access, mathematical analysis, and presentation-quality business graphs. Using COM, a developer may choose a database-access component by one vendor, a business graph component by another, and integrate these into a mathematical analysis package produced by yet a third.

This section covers:

- "Concepts and Terminology" on page 8-2
- "Overview of MATLAB COM Support" on page 8-4

Concepts and Terminology

Here are some terms that you should be familiar with when reading this chapter.

COM Objects

A COM object is an instance of a component object class, or component. COM enforces complete encapsulation of the object, preventing its data and implementation from being accessed directly. The methods of an object must be accessed through *interfaces*. (See "COM Interfaces", below).

COM objects operate either as *in-process* or *out-of-process* servers. When in-process, the objects are implemented in DLL files, and are loaded into the client application's address space. No interprocess communication is required. For out-of-process, the objects are in EXE files, and run in a separate server

Ex.1049 / Page 564 of 748
Intel Corporation et al

process and the application must communicate with them across process boundaries.

Objects may also reside on a remote system connected by a network. In this case, the *Distributed Component Object Model*, or DCOM, is used.

COM Interfaces

An interface provides access to the methods of a COM object. In fact, the only way to access a COM object is through an interface. Through certain methods (`get` and `set`), you also gain access to the object's properties. An interface does not contain any method implementation; it just serves as a pointer to the methods within the object.

Interfaces usually group a set of logically related methods together. An object may, and frequently does, *expose* (or make available) more than one interface. In order to use any COM object, you must learn about which interfaces it supports, and the methods, properties, and events implemented by the object. The COM object's vendor provides this information.

Automation

Automation is a simpler method of communication between COM clients and servers. It uses a single standard COM interface called *IDispatch*. This interface enables the client to find out about and invoke or access methods and properties supported by a COM object. A client and server that communicate using IDispatch are known as an Automation client and Automation server.

IDispatch is the only interface supported by MATLAB. Custom and dual interfaces are not supported.

Controls

A control is an Automation component that has a user interface, enabling it to respond to actions taken by the user. A window that has **OK** and **Cancel** buttons is an example of a control. A control runs in the process address space of its client application. The client, in this case, is said to be a control *container* since it contains the control.

Because a control communicates through Automation, the client application is able to directly invoke its methods and access its properties. The control must also be able to communicate back to the client. This enables it to notify the client application of an event occurence, such as a mouse click.

Ex.1049 / Page 565 of 748
Intel Corporation et al

Overview of MATLAB COM Support

You can configure MATLAB to either control or be controlled by other COM components. When MATLAB controls another component, MATLAB is the *client*, and the other component is the *server*. When MATLAB is controlled by another component, it is acting as the server.

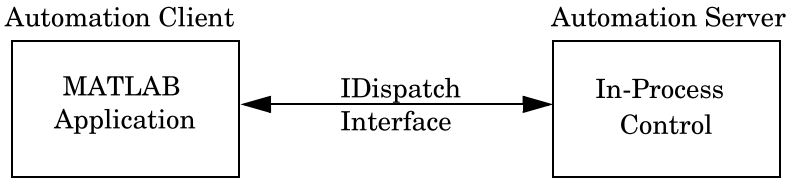
MATLAB supports four different COM client-server configurations:

- "MATLAB Client and In-Process (Control) Server" on page 8-4
- "MATLAB Client and Out-of-Process Server" on page 8-5
- "Client Application and MATLAB Automation Server" on page 8-6
- "Client Application and MATLAB Engine Server" on page 8-6

The first three configurations communicate using an Automation interface that the client program, in turn, accesses via a COM interface called IDispatch. The last MATLAB configuration uses a faster, custom COM interface called IEngine.

MATLAB Client and In-Process (Control) Server

You would set up the configuration shown here when you want to interact with a COM control. Controls are application components that can be both visually and programmatically integrated into a control container, such as a MATLAB figure window. Some examples of useful controls are the Microsoft Internet Explorer Web Browser control, the Microsoft Windows Communications control for serial port access, and the graphical user interface controls delivered with the Visual Basic development environment.



Your MATLAB application is the Automation client. It "contains" the control, because the control runs in the MATLAB process address space rather than as a separate process. The COM control runs as the Automation server.

Ex.1049 / Page 566 of 748
Intel Corporation et al

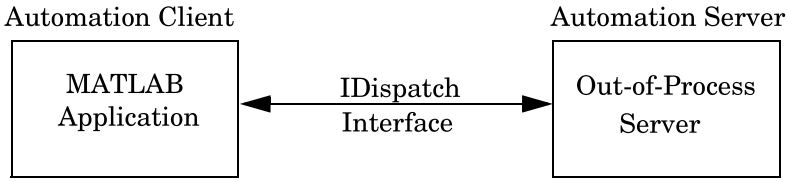
You create the control using the MATLAB `actxcontrol` function. This function creates a MATLAB figure window and positions the graphical user interface of the control within that window. Users commonly interact with this interface using GUI menus, buttons, etc. Doing so often triggers an *event* that gets communicated from the control to its client MATLAB application. The client then decides what to do about the event and responds appropriately.

MATLAB ships with a very simple sample ActiveX control that draws a circle on the screen and displays some text. This allows MATLAB users to try out MATLAB COM control support with a known control. For more information, see “MATLAB Sample Control” on page 8-30.

To learn more about working with MATLAB as a client, see “MATLAB COM Client Support” on page 8-8 and “Additional COM Client Information” on page 8-32

MATLAB Client and Out-of-Process Server

This configuration runs the server in a separate process from that of the client application. The server is no longer contained by the client process, and it does not run as part of MATLAB in a figure window. Any user interface displayed is in a separate window. Examples of local servers are Microsoft Excel and Microsoft Word.



The client and server use Automation to communicate through a COM IDispatch interface. Since the server runs in a separate process, it may be run on either a local or remote system. There is currently no support for events generated from Automation servers.

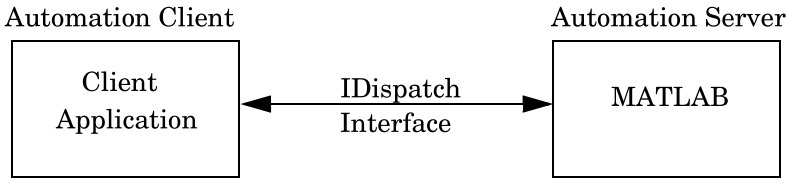
The MATLAB `actxserver` function creates a COM Automation server and returns a COM object for the server’s default interface.

To learn more about working with MATLAB as a client, see “MATLAB COM Client Support” on page 8-8 and “Additional COM Client Information” on page 8-32

Ex.1049 / Page 567 of 748
Intel Corporation et al

Client Application and MATLAB Automation Server

MATLAB operates as the Automation server in this configuration. It can be created and 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.



MATLAB Automation server capabilities include the ability to execute commands in the MATLAB workspace, and to get and put matrices directly from and into the workspace. You can start a MATLAB server to run in either a dedicated or shared mode. You also have the option of running it on a local or remote system.

Note Because VBScript client programs require an Automation interface to communicate with servers, this is the only configuration that will support a VBScript client.

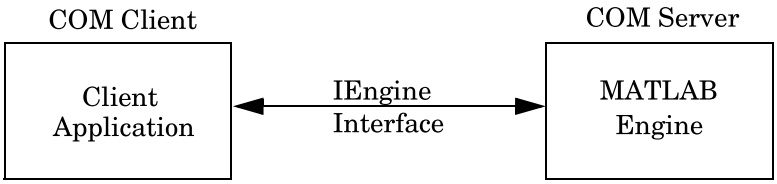
To learn more about working with MATLAB Automation servers, see “MATLAB Automation Server Support” on page 8-36 and “Additional Automation Server Information” on page 8-40.

Client Application and MATLAB Engine Server

MATLAB provides a faster, custom interface called *IEngine* for client applications written in C, C++, or Fortran. MATLAB uses this COM interface to communicate between the client application and the *MATLAB Engine* running as a COM server. A library of functions is provided with MATLAB that

Ex.1049 / Page 568 of 748
Intel Corporation et al

enables you to start and end the server process, and to send commands to be processed by MATLAB.



To learn more about the MATLAB Engine and the functions provided in its C and Fortran libraries, see Chapter 6, “Calling MATLAB from C and Fortran Programs”.

Ex.1049 / Page 569 of 748
Intel Corporation et al

MATLAB COM Client Support

This section introduces the MATLAB functions you need to create, manipulate, and destroy COM control and server objects. These objects are instances of the MATLAB `COM` class. The topics covered here are

- "Creating COM Objects" on page 8-8
- "Object Properties" on page 8-10
- "Invoking Methods" on page 8-17
- "COM Control Events" on page 8-20
- "Identifying Objects" on page 8-23
- "Saving Your Work" on page 8-24
- "Releasing COM Interfaces and Objects" on page 8-25
- "Writing Event Handlers" on page 8-26
- "Examples of MATLAB as an Automation Client" on page 8-30

MATLAB includes three demos showing how to use the COM client interface. To run any of the demos, click on the **Demos** tab in the MATLAB Help Browser. Then click to expand the folder called Automation Client Interface (COM).

Creating COM Objects

To create a COM Automation control or server in MATLAB, use one of the two functions shown here.

Function	Description
`actxcontrol`	Create a COM Automation control
`actxserver`	Create a COM Automation server

Both of these functions require an argument called the `ProgID`. This is a string that is defined by the vendor and can be obtained from the vendor's documentation. For example, the `ProgID` for MATLAB is `matlab.application`.

`actxcontrol` and `actxserver` create an object that is an instance of the MATLAB `com` class. These functions return a handle to the object that you use to reference the object in other COM function calls.

Ex.1049 / Page 570 of 748
Intel Corporation et al

This example creates a COM server application running Excel. The returned handle is assigned to `h`:

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

Getting Interfaces from an Object

Once an object has been created, you can obtain interfaces from it, either by assigning an `Interface` property from the object or by invoking certain methods on it.

This example creates a `Workbooks` interface to the Excel application using the `get` function. The `get` function returns an interface handle that is assigned here to `w`. Use this handle in other COM function calls as a reference to this interface:

```
w = get(h, 'Workbooks')
w =
   Interface.excel.application.Workbooks
```

Repositioning a Control

When MATLAB creates a COM control, it places the control figure inside of a MATLAB figure window. You can create the figure window yourself and specify the control's initial position in the window with the `actxcontrol` function:

```
f = figure ('position', [0 0 300 300]);
sample = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);
```

Once a control has been created, you can change its shape and position in the window with the `move` function.

Function	Description
`move`	Resize or reposition a control in its parent window

Observe what happens to the object in your figure window when you specify new origin coordinates (`70, 120`) and a new height dimension (`100`):

```
move(sample, [70 120 200 100]);
```

Object Properties

Use these MATLAB functions to get, set, and modify the properties of a COM object or interface, or to add your own custom properties.

Function	Description
`addproperty`	Add a custom property to a COM object
`deleteproperty`	Remove a custom property from a COM object
`get`	List one or more properties and their values
`inspect`	Display graphical interface to list and modify property values
`propedit`	Display the control's built-in property page
`set`	Set a property on an interface

Getting the Value of a Property

The `get` function returns information on one or more properties belonging to a COM object or interface. Use `get` with only the handle argument, and MATLAB returns a list of all properties for the object, and their values:

```
h = actxserver('excel.application');

get(h)
ans =
   Application: [1x1 Interface.excel.application.Application]
       Creator: 'xlCreatorCode'
        Parent: [1x1 Interface.excel.application.Parent]
    ActiveCell: []
   ActiveChart: [1x50 char]
 ActivePrinter: '\\PRINTERS\Kunkel on Ne01:'
   ActiveSheet: []
  ActiveWindow: []
         AddIns: [1x1 Interface.excel.application.AddIns]
           :
           :
```

To return the value of just one property, specify the property name in the argument list:

```
get(h, 'OrganizationName')
ans =
   The MathWorks, Inc.
```

Property names are not case sensitive and may also be abbreviated, as long as you include enough letters in the name to make it unambiguous. You can use `'org'` in place of the full `'OrganizationName'` property name used above:

```
get(h, 'org')
ans =
   The MathWorks, Inc.
```

Setting the Value of a Property

Use `set` to change the value of a property, specifying both the property name and new value:

```
set(h, 'DefaultFilePath', 'C:\ExcelWork');
```

You can change more than one property at a time using this syntax:

```
set(h, 'prop1', 'value1', 'prop2', 'value2', ...);
```

For example, to set the `Label` and `Radius` fields of COM object h, use

```
set(h, 'Label', 'Click to fire event', 'Radius', 40);
```

Get and Set on Multiple Objects

You can use the `get` and `set` functions on more than one object at a time by putting the object handles into a vector and then operating on the vector.

This example creates a vector of handles to four Microsoft Calendar objects. It then modifies the Day property of all the objects in one operation by invoking `set` on the vector.

```
h1 = actxcontrol('mscal.calendar', [0 200 250 200]);
h2 = actxcontrol('mscal.calendar', [250 200 250 200]);
h3 = actxcontrol('mscal.calendar', [0 0 250 200]);
h4 = actxcontrol('mscal.calendar', [250 0 250 200]);
H = [h1 h2 h3 h4];
```

```
set(H, 'Day', 23)
get(H, 'Day')
ans =
    [23]
    [23]
    [23]
    [23]
```

Using Enumerated Values for Properties

Enumeration makes examining and changing properties easier because each possible value for the property is given a string to represent it. For example, one of the values for the `DefaultSaveFormat` property in an Excel application is `xlUnicodeText`. This is much easier to remember than a numeric value like `57`.

Finding All Enumerated Properties. The MATLAB COM `get` and `set` functions support enumerated values for properties for those applications that provide them. To see which properties use enumerated types, use the `set` function with just the object handle argument:

```
h = actxserver('excel.application');

set(h)
ans =
                     Creator: {'xlCreatorCode'}
           ConstrainNumeric: {}
       CopyObjectsWithCells: {}
                     Cursor: {4x1 cell}
                CutCopyMode: {2x1 cell}
                        .
                        .
```

MATLAB displays the properties that accept enumerated types as nonempty cell arrays. Properties for which there is a choice of settings are displayed as a multirow cell array, with one row per setting (see `Cursor` in the example above). Properties for which there is only one possible setting are displayed as a one row cell array (see `Creator`, above).

Ex.1049 / Page 574 of 748
Intel Corporation et al

To display the current values of these properties, use `get` with just the object handle argument:

```
get(h)
                      Creator: 'xlCreatorCode'
            ConstrainNumeric: 0
        CopyObjectsWithCells: 1
                      Cursor: 'xlDefault'
                CutCopyMode: ''
                        .
                        .
```

Setting an Enumerated Value. To list all possible enumerated values for a specific property, use `set` with the property name argument. The output is a cell array of strings, one string for each possible setting of the specified property:

```
set(h, 'Cursor')
ans =
    'xlIBeam'
    'xlDefault'
    'xlNorthwestArrow'
    'xlWait'
```

To set the value of a property to an enumerated type, use `set` with the property name and enumerated value. (You can also set these properties to an integer value.) Note that you can abbreviate the enumeration string, as in the second line shown below. You must use enough letters in the string to make it unambiguous. The enumeration string is also case-insensitive.

Either of the following commands sets the `Cursor` to a `NorthwestArrow` type:

```
set(h, 'Cursor', 'xlNorthwestArrow')
set(h, 'Cursor', 'xln')
```

Use `get` with the property name argument to read the value of the property you have just set:

```
get(h, 'Cursor')
ans =
   xlNorthwestArrow
```

Using the Property Inspector

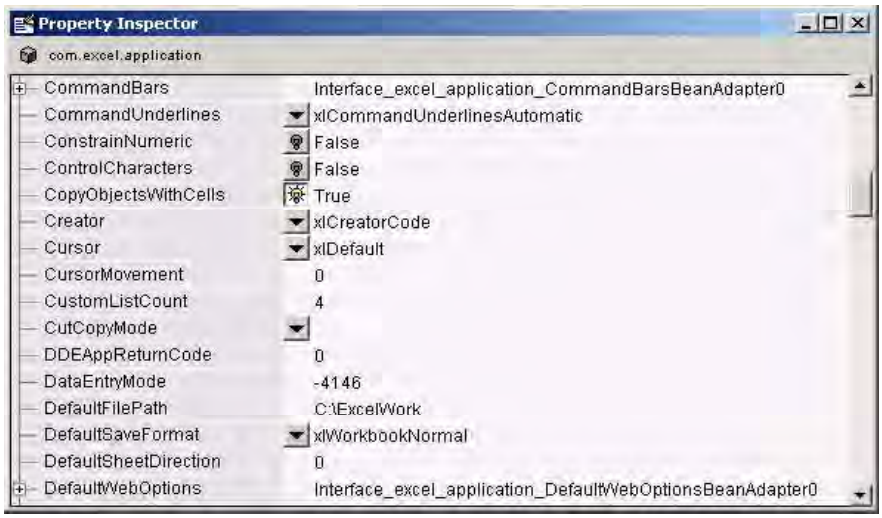
MATLAB also provides a graphical user interface to display and modify properties. You can open the Property Inspector by either of these two methods:

- Invoke the `inspect` function from the MATLAB command line
- Double-click on the object in the Workspace Browser

Using the same Excel server, invoke the `inspect` function to display a new window showing the server object's properties:

```
inspect(h)
```

Scroll down until you see the `DefaultFilePath` property that you just changed. It should read `C:\ExcelWork`.



Using the Property Inspector, change `DefaultFilePath` once more, this time to `C:\MyWorkDirectory`. To do this, click on the property name at the left and then enter the new value at the right.

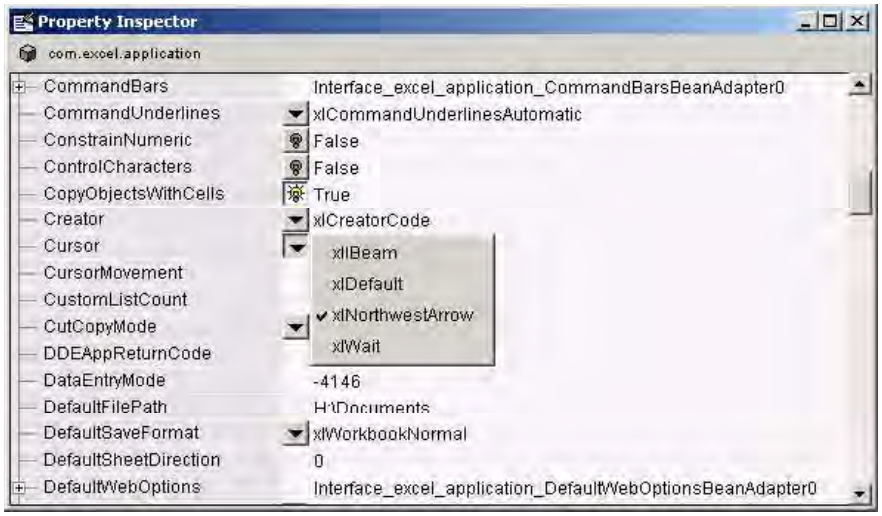
Ex.1049 / Page 576 of 748
Intel Corporation et al

Now check this field in the MATLAB command window and confirm that it has changed:

```
get(h, 'DefaultFilePath')
ans =
   C:\MyWorkDirectory
```

Note If you modify properties at the MATLAB command line, you must refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector by reinvoking `inspect` on the object.

Using the Property Inspector on Enumerated Values. Properties that accept enumerated values are marked by a button in the Property Inspector window. The window shown below displays four enumerated values for the `Cursor` property. The current value, `xlNorthwestArrow`, is indicated by a checkmark.



To change a property's value using the Property Inspector, simply click on the button to display the options for that property, and then click on the desired value.

Custom Properties

You can attach your own properties to a control using the `addproperty` function. The syntax shown here creates a custom property for control, `h`:

```
addproperty(h, 'propertyName')
```

This example creates the `mwsamp` control, adds a new property called `Position` to it, and assigns the value `[200 120]` to that property:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [200 120 200 200]);
addproperty(h, 'Position');
set(h, 'Position', [200 120]);
```

Use `get` to list all properties of control, `h`. You see that the new `Position` property has been added:

```
get(h)
ans =
     Label: 'Label'
    Radius: 20
  Position: [200 120]

get(h, 'Position')
ans =
   200   120
```

To remove custom properties from a control, use `deleteproperty` with the following syntax:

```
deleteproperty(h, 'propertyName')
```

For example, delete the `Position` property that you just created, and use `get` to show that it no longer exists:

```
deleteproperty(h, 'Position');

get(h)
     Label: 'Label'
    Radius: 20
```

Invoking Methods

Use these MATLAB functions to find out what methods a COM object has and to invoke any COM or MATLAB method on that object.

Function	Description
`invoke`	Invoke a method or display a list of methods and types
`methods`	List all method names for the control or server
`methodsview`	GUI interface to list information on all methods and types

The `invoke` function either invokes a method or returns information on the methods implemented for a COM object or available through an interface. Create a Microsoft Calendar control and use `invoke` without arguments to return a list of all methods for the control:

```
cal = actxcontrol('mscal.calendar', [0 0 500 500])
cal =
   COM.mscal.calendar

invoke(cal)
   NextDay = HRESULT NextDay(handle)
   NextMonth = HRESULT NextMonth(handle)
   NextWeek = HRESULT NextWeek(handle)
   NextYear = HRESULT NextYear(handle)
   PreviousDay = HRESULT PreviousDay(handle)
   PreviousMonth = HRESULT PreviousMonth(handle)
   PreviousWeek = HRESULT PreviousWeek(handle)
   PreviousYear = HRESULT PreviousYear(handle)
   Refresh = HRESULT Refresh(handle)
   Today = HRESULT Today(handle)
   AboutBox = HRESULT AboutBox(handle)
```

You can invoke any of these methods on the object using the arguments shown in the returned method list. The following example reads today's date and then advances it by 50 years using the `NextYear` method in a loop.

To get today's date, type

```
sprintf('%d/%d/%d', cal.Month, cal.Day, cal.Year)
ans =
   11/5/2001
```

Now, use `NextYear` to advance the date, and then verify the results:

```
for k=1:50
   NextYear(cal);
end

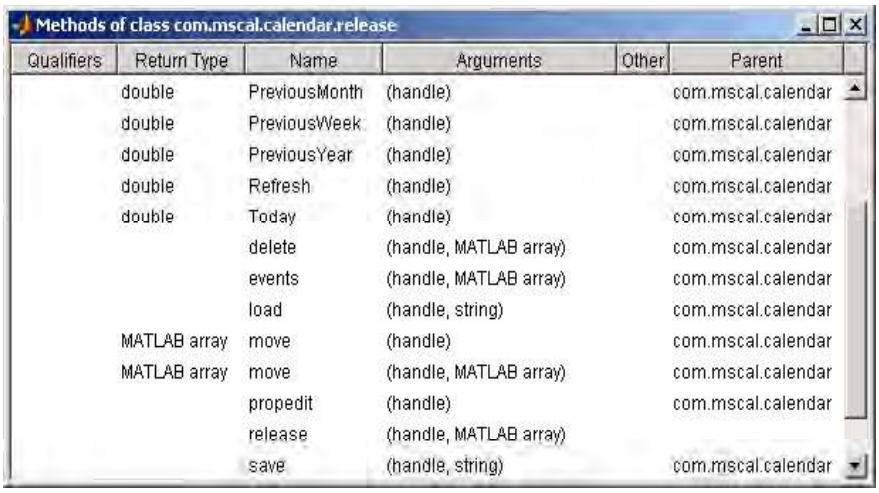
sprintf('%d/%d/%d', cal.Month, cal.Day, cal.Year)
ans =
   11/5/2051
```

Graphical Methods Display

MATLAB also has a graphical user interface to view the methods of an object along with related fields of information. Type

```
methodsview(h)
```

to bring up a window such as the one shown here.



Methods that return void show no Return Type in the display.

Returning Multiple Output Arguments

If you know that a server function supports multiple outputs, you can return any or all of those outputs to a MATLAB client. Specify the output arguments within brackets on the left-hand side of the equation. This gives the MATLAB client code access to any values returned by the server function.

The syntax shown here shows a server function being called by the MATLAB client. The function's return value is shown as `retval`. The function's output arguments (`out1`, `out2`, `...`) follow this.

```
[retval out1 out2 ...] = functionname(handle, in1, in2, ...);
```

MATLAB makes use of the pass by reference capabilities in COM to implement this feature. Note that pass by reference is a COM feature. It is not available in MATLAB at this time.

Argument Callouts in Error Messages

When a MATLAB client sends a command with an invalid argument to a COM server application, the server sends back an error message similar to that shown here, identifying the invalid argument. Be careful when interpreting the argument callout in this type of message.

```
PutFullMatrix(handle, 'a', 'base', 7, [5 8]);
??? Error: Type mismatch, argument 3.
```

In the `PutFullMatrix` command shown above, it is the fourth argument, `7`, that is invalid. (It is scalar and not the expected array data type.) However, the error message identifies the failing argument as `argument 3`.

This is because the COM server receives only the last four of the arguments shown in the MATLAB code. (The `handle` argument merely identifies the server. It does not get passed to the server). So the server sees `'a'` as the first argument, and the invalid argument, `7`, as the third.

As another example, submitting the same command with the `invoke` function makes the invalid argument fifth in the MATLAB client code. Yet the server still identifies it as `argument 3` because neither of the first two arguments are seen by the server.

```
invoke(handle, 'PutFullMatrix', 'a', 'base', 7, [5 8]);
??? Error: Type mismatch, argument 3.
```

COM Control Events

Use these MATLAB functions on a control to find out what events the control generates, and to enable or disable handler routines to respond to those events.

Function	Description
`eventlisteners`	Return a list of events attached to listeners
`events`	List all events registered for a control
`registerevent`	Register an event handler with a control's event
`unregisterallevents`	Unregister all events for a control
`unregisterevent`	Unregister an event handler with a control's event

COM controls listen to events initiated by users of the control. An example of an event is clicking a mouse button in a window. If a handler routine for the event has been written and registered with the control, then the control responds to the event by executing the event handler routine.

Registering Events with the Control

You can register event handler routines with their associated events either when you create the control, using `actxcontrol`, or any time afterwards, using the `registerevent` function.

This example registers two events (`Click` and `MouseDown`) and two respective handler routines (`myclick` and `mymoused`) with the `mwsamp` control at the time the control is created. Event handlers for the `mwsamp` control are defined in the section, "Sample Event Handlers" on page 8-28:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f, ...
    {'Click' 'myclick'; 'MouseDown' 'mymoused'});
```

You could have also used the following `registerevent` call at any time after creating the control:

```
registerevent(h, {'Click' 'myclick'; 'MouseDown' 'mymoused'});
```

Listing a Control's Events

Use the `events` function to list all events the control is capable of responding to. Note that `events` lists both registered and unregistered events. In this example, the `DblClick` event is not registered at this time:

```
events(h)
   Click = void Click()
   DblClick = void DblClick()
   MouseDown = void MouseDown(int16 Button, int16 Shift,
      Variant x, Variant y)
```

To list only those events that are currently registered with the control, use the `eventlisteners` function. This function lists both the event names and their handler routines:

```
eventlisteners(h)
ans =
    'click'        'myclick'
    'mousedown'    'mymoused'
```

Firing an Event

When MATLAB creates the `mwsamp` control, it also displays a figure window showing a label and circle at the center. If you click on different positions in this window, you will see a report of the `X` and `Y` position of the mouse in the MATLAB command window. MATLAB is responding to the `MouseDown` event:

```
The X position is:
ans =
    [122]
The Y position is:
ans =
    [63]
```

You also see the following line reported in response to the `Click` event:

```
Single click function
```

Double-clicking the mouse does nothing different, since the `DblClick` event has yet to be registered.

Ex.1049 / Page 583 of 748
Intel Corporation et al

Unregistering Events

Use the `unregisterevent` function to unregister an event handler with an event. For example, if you unregister the `MouseDown` event, you will notice that MATLAB no longer reports the `X` and `Y` position when you click in the window:

```
unregisterevent(h, {'MouseDown' 'mymoused'});
```

Now, register the `DblClick` event, connecting it with handler routine `my2click`:

```
registerevent(h, {'DblClick', 'my2click'});
```

If you call `eventlisteners` again, you will see that the registered events are now `Click` and `DblClick`:

```
eventlisteners(h)
ans =
    'click'       'myclick'
    'dblclick'    'my2click'
```

When you double-click the mouse button, MATLAB responds to both the `Click` and `DblClick` events by displaying the following in the command window:

```
Single click function
Double click function
```

An easy way to unregister all events for a control is to use the `unregisterallevents` function. When there are no events registered, `eventlisteners` returns an empty cell array:

```
unregisterallevents(h)

eventlisteners(h)
ans =
     {}
```

Clicking the mouse in the control window now does nothing since there are no active events.

Identifying Objects

Use these MATLAB functions to get information about a COM object.

Function	Description
`class`	Return the class of a COM object
`isa`	Detect a COM object of a given class
`isevent`	Determine if an item is an event of a COM object
`ismethod`	Determine if an item is a method of a COM object
`isprop`	Determine if an item is a property of a COM object

Create a COM object, `h`, in an Automation server running Excel, and also a `Workbooks` interface, `w`, to the object:

```
h = actxserver('excel.application');
w = get(h, 'Workbooks');
```

To find out the class of variable `w`, use the `class` function:

```
class(w)
ans =
   Interface.excel.application.Workbooks
```

To test a variable against a known class name, use `isa`:

```
isa(h, 'COM.excel.application')
ans =
     1
```

To see if `UsableWidth` is a property of object `h`, use `isprop`:

```
isprop(h, 'UsableWidth')
ans =
     1
```

To see if SaveWorkspace is a method of object h, use ismethod. Method names are case sensitive and cannot be abbreviated:

```
ismethod(h, 'SaveWorkspace')
ans =
     1
```

Create the sample mwsamp control that comes with MATLAB, and use isevent to see if DblClick is one of the events that this control recognizes:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

isevent(h, 'DblClick')
ans =
     1
```

Saving Your Work

Use these MATLAB functions to save and restore the state of a COM control object.

Function	Description
load	Initialize a COM control object from a file
save	Serialize a COM control object to a file

Save the current state of a COM control to a file using the save function. The following example creates an mwsamp control and saves its original state to the file mwsample:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
save(h, 'mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```
set(h, 'Label', 'Circle');
set(h, 'Radius', 50);
Redraw(h);
```

Using the `load` function, you can restore the control to its original state:

```
load(h, 'mwsample');
get(h)
ans =
     Label: 'Label'
    Radius: 20
```

Note The COM `save` and `load` functions are only supported for controls at this time.

Releasing COM Interfaces and Objects

Use these MATLAB functions to release or delete a COM object or interface.

Function	Description
`delete`	Delete a COM object or interface
`release`	Release a COM object or interface

When each interface is no longer needed, use the `release` function to release the interface and reclaim the memory used by it. When the entire control or server is no longer needed, use the `delete` function to delete it. Alternatively, you can use the `delete` function on any valid interface. All interfaces for that object are automatically released and the server or control itself is deleted.

Note In versions of MATLAB earlier than 6.5, failure to explicitly release interface handles or delete the control or server often results in a memory leak. This is true even if the variable representing the interface or COM object has been reassigned. In MATLAB 6.5 and later, the control or server, along with all interfaces to it, is destroyed on reassignment of the variable or when the variable representing a COM object or interface goes out of scope.

MATLAB automatically releases all interfaces for a control when the figure window that contains that control is deleted or closed. MATLAB also automatically releases all handles for an Automation server when MATLAB is shut down.

Writing Event Handlers

An event is fired when a control wants to notify its container that something of interest has occurred. For example, many controls trigger an event when the user single-clicks on the control. In MATLAB, you can create and register your own M-file functions so that they respond to events when they occur. These functions serve as event handlers. You can create one handler function to handle all events or a separate handler for each type of event.

Register your handler functions either at the time you create the control (using `actxcontrol`), or at any time afterwards (using `registerevent`). Specify the event handler in the argument list, as shown below for `actxcontrol`. The event handler argument can be either the name of a single callback routine or a cell array that associates specific events with their respective event handlers:

```
h = actxcontrol (progid, position, handle, ...
callback | {event1 eventhandler1; event2 eventhandler2; ...})
```

When you specify the single `callback` routine, MATLAB registers all events with that one routine. When any event is fired, MATLAB executes the common `callback` routine.

You can list all the events that a COM object recognizes using the `events` function. For example, to list all events for the `mwsamp` control, use

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

events(h)
   Click = void Click()
   DblClick = void DblClick()
   MouseDown = void MouseDown(int16 Button, int16 Shift,
      Variant x, Variant y)
```

Arguments Passed to Event Handlers

When a registered event is triggered, MATLAB passes information from the event to its handler function as shown in this table.

Arguments Passed by MATLAB

Arg. No.	Contents	Format
`1`	Object Name	MATLAB `com` class
`2`	Event ID	`double`
`3`	Start of Event Arg. List	As passed by the control
`end-2`	End of Event Arg. List (Arg. N)	As passed by the control
`end-1`	Event Structure	`structure`
`end`	Event Name	`char array`

When writing an event handler function, use the Event Name argument to identify the source of the event. Get the arguments passed by the control from the Event Argument List (arguments `3` through `end-2`). All event handlers must accept a variable number of arguments:

```
function event (varargin)
if (varargin{end}) == 'MouseDown')      % Check the event name
    x_pos = varargin{5};              % Read 5th Event Argument
    y_pos = varargin{6};              % Read 6th Event Argument
end
```

Note The values passed vary with the particular event and control being used.

Event Structure

The second to last argument passed by MATLAB is the Event Structure, which has the following fields.

Fields of the Event Structure

Field Name	Description	Format
`Type`	Event Name	`char array`
`Source`	Control Name	MATLAB `com` class
`EventID`	Event Identifier	`double`
`Event Arg Name 1`	Event Arg Value 1	As passed by the control
`Event Arg Name 2`	Event Arg Value 2	As passed by the control
etc.		

For example, when the `MouseDown` event of the `mwsamp` control is triggered, MATLAB passes this Event Structure to the registered event handler:

```
    Type: 'MouseDown'
  Source: [1x1 COM.mwsamp.mwsampctrl.2]
 EventID: -605
  Button: 1
   Shift: 0
       x: 27
       y: 24
```

Sample Event Handlers

Specify a single callback, `sampev`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
    gcf, 'sampev')
h =
    COM.mwsamp.mwsampctrl.2
```

Or specify several events using the cell array format:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f, ...
    {'Click' 'myclick'; 'DblClick' 'my2click'; ...
    'MouseDown' 'mymoused'});
```

The event handlers, `myclick.m`, `my2click.m`, and `mymoused.m`, are

```
function myclick(varargin)
disp('Single click function')

function my2click(varargin)
disp('Double click function')

function mymoused(varargin)
disp('You have reached the mouse down function')
disp('The X position is: ')
double(varargin{6})
disp('The Y position is: ')
double(varargin{7})
```

Alternatively, you can use the same event handler for all the events you want to monitor using the cell array pairs. Response time will be better than using the callback style.

For example:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', ...
[0 0 200 200], f, {'Click' 'allevents'; ...
'DblClick' 'allevents'; 'MouseDown' 'allevents'})
```

where `allevents.m` is

```
function allevents(varargin)
if (varargin{3} = 'Click')
   disp ('Single Click Event Fired')
elseif (varargin{3} = 'DblClick')
   disp ('Double Click Event Fired')
elseif (varargin{3} = 'MouseDown')
   disp ('Mousedown Event Fired')
end
```

Examples of MATLAB as an Automation Client

This section provides examples of using MATLAB as an Automation client with controls and servers.

MATLAB Sample Control

MATLAB ships with a very simple example COM control that draws a circle on the screen, displays some text, and fires events when the user single- or double-clicks on the control. Create the control by running the `mwsamp.m` file in the directory, `winfun\comcli`, or type

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 300 300]);
```

This control is stored in the MATLAB `bin`, or executable, directory along with the control's *type library*. The type library is a binary file used by COM tools to decipher the control's capabilities. See the section "Writing Event Handlers" on page 8-26 for other examples that use the `mwsamp` controls.

Using MATLAB as an Automation Client

This example uses MATLAB as an Automation client and Microsoft Excel as the server. It provides a good overview of typical functions. In addition, it is a good example of using the Automation interface of another application:

```
% MATLAB Automation client example
%
% Open Excel, add workbook, change active worksheet,
% get/put array, save.

% First, open an Excel Server.
e = actxserver('excel.application');

% Insert a new workbook.
eWorkbooks = get(e, 'Workbooks');
eWorkbook = Add(eWorkbooks);
set(e, 'Visible', 1);

% Make the second sheet active.
eActiveWorkbook = get(e, 'ActiveWorkBook');
eSheets = get(eActiveWorkbook, 'Sheets');
eSheet2 = Item(eSheets, 2);
Activate(eSheet2);
```


Ex.1049 / Page 592 of 748
Intel Corporation et al

```
% Get a handle to the active sheet.
eActiveSheet = get(e, 'ActiveSheet');

% Put a MATLAB array into Excel.
A = [1 2; 3 4];
eActiveSheetRange = Range(eActiveSheet, 'A1', 'B2');
set(eActiveSheetRange, 'Value', A);

% Get back a range.  It will be a cell array, since the cell range
% can contain different types of data.
eRange = Range(eActiveSheet, 'A1', 'B2');
B = get(eRange, 'Value');

% Convert to a double matrix.  The cell array must contain only
% scalars.
B = reshape([B{:}], size(B));

% Now, save the workbook.
SaveAs(eWorkbook, 'myfile.xls');

% To avoid saving the workbook and being prompted to do so,
% uncomment the following code.
% set(eWorkbook, 'Saved', 1);
% Close(eWorkbook);

% Quit Excel and delete the server.
% Quit(e);
% delete(e);
```

Note Make sure that you always close any workbooks that you add in Excel. This can prevent potential memory leaks.

Additional COM Client Information

Using COM Collections

COM *collections* are a way to support groups of related COM objects that can be iterated over. A collection is itself a special interface with a `Count` property (read only), which contains the number of items in the collection, and an `Item` method, which allows you to retrieve a single item from the collection.

The `Item` method is indexed, which means that it requires an argument that specifies which item in the collection is being requested. The data type of the index can be any data type that is appropriate for the particular collection and is specific to the control or server that supports the collection. Although integer indices are common, the index could just as easily be a string value. Often, the return value from the `Item` method is itself an interface. Like all interfaces, this interface should be released when you are finished with it.

This example iterates through the members of a collection. Each member of the collection is itself an interface (called `Plot` and represented by a MATLAB COM object called `hPlot`.) In particular, this example iterates through a collection of `Plot` interfaces, invokes the `Redraw` method for each interface, and then releases each interface:

```
hCollection = get(hControl, 'Plots');
for i = 1:get(hCollection, 'Count')
   hPlot = invoke(hCollection, 'Item', i);
   invoke(hPlot, 'Redraw');
   release(hPlot);
end;
release(hCollection);
```