MATLAB® 7
External Interfaces
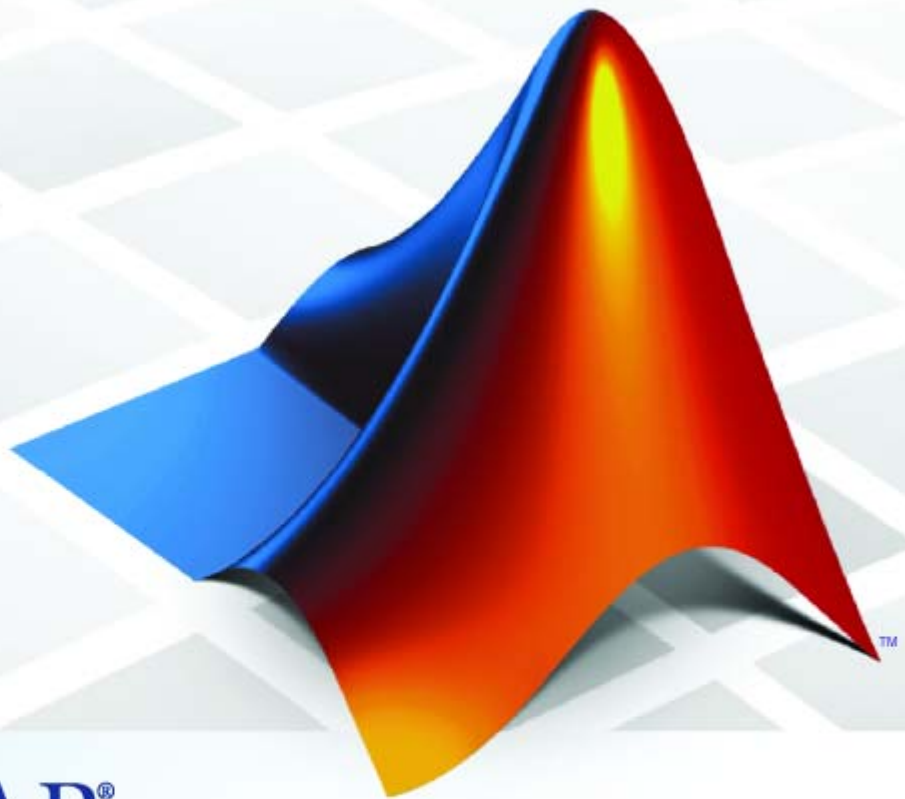
MATLAB®

How to Contact The MathWorks

www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support

suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB® External Interfaces

Trademarks

Patents

Contents

Calling C and Fortran Programs from MATLAB Command Line

3

Creating C Language MEX-Files

4

Creating Fortran MEX-Files

5

Calling MATLAB Software from C and Fortran Programs

6

Using Sun Java Classes in MATLAB Software

7

COM Support for MATLAB Software

8

MATLAB COM Client Support

9

MATLAB COM Automation Server Support

10

11

Web Services in MATLAB Applications

12

Serial Port I/O

Examples

A

Index

1

Importing and Exporting Data

Using MAT-Files

In this section...

Introduction

MAT-files, the data file format MATLAB® software uses for saving data to your disk, provide a convenient mechanism for moving data between platforms and for importing and exporting data to stand alone MATLAB applications.

To simplify your use of MAT-files in applications outside of the MATLAB environment, we have developed a library of access routines with a `mat` prefix that you can use in your own C or Fortran programs to read and write MAT-files. Programs that access MAT-files also use the `mx` prefixed API (application program interface) routines discussed in Chapter 4, "Creating C Language MEX-Files" and Chapter 5, "Creating Fortran MEX-Files".

Importing Data into the MATLAB Workspace

The best method for importing data into MATLAB depends on how much data there is, whether the data is already in machine-readable form, and what format the data is in. Here are some choices; select the one that best meets your needs.

- **Enter the data at the MATLAB command prompt.**

  For small amounts of data, less than 10-15 elements, type the data at the command prompt using brackets `[]`. This method is awkward for large amounts of data because you can't edit your input.

- **Create data in an M-file.**

  With a text editor, create an M-file to enter data as an explicit list of elements. Use this method when the data is not already in computer-readable format and must be typed in. Use the editor to change the data or correct mistakes, then rerun the M-file to reenter the data.

- **Load data from an ASCII flat file.**

  A *flat file* stores data in ASCII format, with fixed-length rows terminated by new lines (carriage returns) and with spaces separating the numbers. Edit ASCII flat files using a text editor and use the `load` command to read them directly into the workspace. MATLAB creates a variable with the same name as the file name.

- **Read data using MATLAB I/O functions.**

  Use `fopen`, `fread`, and other low-level MATLAB I/O functions to read data. This method allows you to load data files from applications that have their own file formats.

- **Write a MEX-file to read the data.**

  Use this method when you have subroutines for reading data files from other applications. For more information, see "Using MEX-Files to Call C and Fortran Programs" on page 3-2.

- **Write a program to translate your data.**

  Write programs in C or Fortran to translate your data into MAT-file format. Use the `load` command to read the MAT-file into the workspace. Refer to the section, "Reading and Writing MAT-Files" on page 1-5, for more information.

Exporting Data from the MATLAB Workspace

There are several methods for exporting MATLAB data.

- **Create a diary file.**

  For small matrices, use the `diary` command to create a diary file, a log of keyboard input and the resulting text output. You can use a text editor to modify the file. The diary file displays the variables and includes the MATLAB commands used during the session, which can be used in documents and reports.

- **Use the `save` command.**

  Save data in ASCII format using the `save` command with the `-ascii` option. For example:

  ```
  A = rand(4,3);
  save temp.dat A -ascii
  ```

  creates an ASCII file called `temp.dat`, which may look something like this:

  ```
  1.3889088e-001   2.7218792e-001   4.4509643e-001
  2.0276522e-001   1.9881427e-001   9.3181458e-001
  1.9872174e-001   1.5273927e-002   4.6599434e-001
  6.0379248e-001   7.4678568e-001   4.1864947e-001
  ```

  The `-ascii` option supports two-dimensional double and character arrays only. Multidimensional arrays, cell arrays, and structures are not supported.

- **Use MATLAB I/O functions.**

  Write the data in a special format using `fopen`, `fwrite`, and other low-level file I/O functions. Use this method for writing data files in file formats required by other applications. For more information, see "Using Low-Level File I/O Functions".

- **Create a MEX-file to write the data.**

  Use this method if you have subroutines for writing data files in the form needed by other applications. For more information, see "Using MEX-Files to Call C and Fortran Programs" on page 3-2.

- **Translate data from a MAT-file.**

  Write data into a MAT-file using the `save` command, then write a program in C or Fortran to translate the MAT-file into your own special format. For more information, see "Reading and Writing MAT-Files" on page 1-5.

Exchanging Data Files Between Platforms

You can work with MATLAB software on different computer systems and send MATLAB applications to users on other systems. MATLAB applications consist of M-files containing functions and scripts, and MAT-files containing binary data.

Both types of files can be transported directly between machines: M-files because they are platform independent and MAT-files because they contain a machine signature in the file header. MATLAB checks the signature when it loads a file and, if a signature indicates that a file is foreign, performs the necessary conversion.

Using MATLAB across different machine architectures requires a facility for exchanging both binary and ASCII data between the machines. Examples of this type of facility include FTP, NFS, and Kermit. When using these programs, be careful to transmit MAT-files in *binary file mode* and M-files in *ASCII file mode*. Failure to set these modes correctly corrupts the data.

Reading and Writing MAT-Files

Use the `save` function to save MATLAB arrays currently in memory to a binary file called a MAT-file. MAT-files have the extension `.mat`. The `load` command reads MATLAB arrays from a MAT-file on disk back into the MATLAB workspace.

A MAT-file contains one or more of the data types supported in MATLAB version 5 or later, including strings, matrices, multidimensional arrays, structures, and cell arrays. MATLAB writes the data sequentially onto disk in a continuous byte stream.

MAT-File Interface Library

The MAT-file interface library contains routines for reading and writing MAT-files. You can call these routines from your own C and Fortran programs. Use these routines, rather than attempt to write your own code, to perform these operations, since using the library insulates your applications from future changes to the MAT-file structure.

Functions in the MAT-file library begin with the three-letter prefix `mat`. These tables list and describe the MAT-functions.

C MAT-File Routines

MAT-Function	Purpose
`matOpen`	Open a MAT-file.

C MAT-File Routines (Continued)

MAT-Function	Purpose
matClose	Close a MAT-file.
matGetDir	Get a list of MATLAB arrays from a MAT-file.
matGetFp	Get an ANSI® C file pointer to a MAT-file.
matGetVariable	Read a MATLAB array from a MAT-file.
matPutVariable	Write a MATLAB array to a MAT-file.
matGetNextVariable	Read the next MATLAB array from a MAT-file.
matDeleteVariable	Remove a MATLAB array from a MAT-file.
matPutVariableAsGlobal	Put a MATLAB array into a MAT-file such that the load command places it into the global workspace.
matGetVariableInfo	Load a MATLAB array header from a MAT-file (no data).
matGetNextVariableInfo	Load the next MATLAB array header from a MAT-file (no data).

Fortran MAT-File Routines

MAT-Function	Purpose
matOpen	Open a MAT-file.
matClose	Close a MAT-file.
matGetDir	Get a list of MATLAB arrays from a MAT-file.
matGetVariable	Get a named MATLAB array from a MAT-file.
matGetVariableInfo	Get header for named MATLAB array from a MAT-file.

Fortran MAT-File Routines (Continued)

MAT-Function	Purpose
matPutVariable	Put a MATLAB array into a MAT-file.
matPutVariableAsGlobal	Put a MATLAB array into a MAT-file.
matGetNextVariable	Get the next sequential MATLAB array from a MAT-file.
matGetNextVariableInfo	Get header for next sequential MATLAB array from a MAT-file.
matDeleteVariable	Remove a MATLAB array from a MAT-file.

Writing Character Data

By default, MATLAB writes character data to MAT-files using Unicode®
character encoding. To override this setting and use your system's default
encoding instead, do one of the following:

- From the command line or a MATLAB function, save your data to the
  MAT-file using the command save -nounicode.

- From a C mex file, open the MAT-file you will write the data to using the
  command matOpen -wL.

See the individual reference pages for these functions for more information.

You can also set a save preference for all MATLAB sessions. For more
information, see "MAT-Files Preferences" in the "General Preferences for
MATLAB" section of the Desktop Tools and Development Environment
documentation.

ASCII Data Formats

When writing character data using Unicode character encoding (the default),
MATLAB checks if the data is 7-bit ASCII. If it is, MATLAB writes the 7-bit
ASCII character data to the MAT-file using 8 bits per character (UTF-8
format), thus minimizing the size of the resulting file. Any character data
that is not 7-bit ASCII is written in 16-bit Unicode form (UTF-16). This
algorithm operates on a per-string basis.

> **Note** Level 4 MAT-files support only ASCII character data. You cannot write a Level 4 MAT-file containing non-ASCII character data. If you create a Level 4 MAT-file with such character data, the original representation of the characters is not preserved.

Converting Character Data

Writing character data to MAT-files using Unicode character encoding enables you to share data with users that have systems with a different default system character encoding scheme, without character data loss or corruption. Although conversion between Unicode encoding and other encoding forms is often lossless, there are scenarios in which round-trip conversions can result in loss of data. The following guidelines may reduce your chances of data loss or corruption.

In order to prevent loss or corruption of character data, all users sharing the data must have at least one of the following in common:

- They exchange Unicode-based MAT-files, and use a version of MATLAB that supports these files.

- Their computer systems all use the same default encoding, and all character data in the MAT-file was written using the `-nounicode` option

For example, if one user on a Japanese language operating system writes ASCII data having more than 7 bits per character to a MAT-file, another user running MATLAB version 6.5 on an English language operating system will be unable to read the data accurately. However, if both have MATLAB version 7, the information can be shared without corruption or loss of data.

Finding Associated Files

A collection of files associated with reading and writing MAT-files is located on your disk. The following table, MAT-Function Subdirectories, lists the path to the required subdirectories for importing and exporting data using MAT-functions. The term *matlabroot* refers to the root directory of your MATLAB installation.

MAT-Function Subdirectories

Platform	Contents	Directories
Microsoft® Windows®	Include files	*matlabroot*\extern\include
	Libraries	*matlabroot*\bin\win32 or *matlabroot*\bin\win64
	Examples	*matlabroot*\extern\examples\eng_mat
UNIX®[1]	Include files	*matlabroot*/extern/include
	Libraries	*matlabroot*/bin/$arch
	Examples	*matlabroot*/extern/examples/eng_mat

Include Files

The include subdirectory holds header files containing function declarations with prototypes for the routines that you can access in the API Library. These files are the same for both Windows and UNIX systems. The subdirectory contains:

- The matrix.h header file that defines MATLAB array access and creation methods
- The mat.h header file that defines MAT-file access and creation methods

Libraries

The libraries subdirectory, that contains shared (dynamically linkable) libraries for linking your programs, is platform dependent.

Shared Libraries on Windows Systems. The bin subdirectory contains the shared libraries for linking your programs:

- The libmat.dll library of MAT-file routines (C and Fortran)
- The libmx.dll library of array access and creation routines

---

1. UNIX is a registered trademark of The Open Group in the United States and other countries.

Shared Libraries on UNIX Systems. The bin/$arch subdirectory, where $arch is your machine's architecture, contains the shared libraries for linking your programs. For example, on Sun™ Solaris™ systems, the subdirectory is bin/sol64:

- The libmat.so library of MAT-file routines (C and Fortran)
- The libmx.so library of array access and creation routines

Example Files

The examples/eng_mat subdirectory contains C and Fortran source code for a number of example files that demonstrate how to use the MAT-file routines. The source code files are the same for both Windows and UNIX systems.

C and Fortran Examples

Library	Description
matcreat.c	C program that demonstrates how to use the library routines to create a MAT-file that can be loaded into MATLAB
matcreat.cpp	C++ version of the matcreat.c program
matdgns.c	C program that demonstrates how to use the library routines to read and diagnose a MAT-file
matdemo1.F	Fortran program that demonstrates how to call the MATLAB MAT-file functions from a Fortran program
matdemo2.F	Fortran program that demonstrates how to use the library routines to read the MAT-file created by matdemo1.F and describe its contents

For more information about MATLAB API files and directories, see "Additional Information for Using MEX-Files" on page 3-61.

Examples of MAT-Files

In this section...
"Creating a MAT-File in C" on page 1-11
"Creating a MAT-File in C++" on page 1-12
"Reading a MAT-File in C" on page 1-12
"Creating a MAT-File in Fortran" on page 1-13
"Reading a MAT-File in Fortran" on page 1-13

Creating a MAT-File in C

The program, `matcreat.c`, illustrates how to use the library routines to create a MAT-file that can be loaded into the MATLAB workspace. The program also demonstrates how to check the return values of MAT-function calls for read or write failures. To see the code, you can open the file in the MATLAB Editor.

To produce an executable version of this program, compile the file and link it with the appropriate library. For details on how to compile and link MAT-file programs on various platforms, see "Compiling and Linking MAT-File Programs" on page 1-15.

Once you have compiled and linked your MAT-file program, you can run the stand alone application you have just produced. This program creates `mattest.mat`, a MAT-file that can be loaded into MATLAB. To run the application, depending on your platform, either double-click its icon or enter `matcreat` at the system prompt:

```
matcreat
Creating file mattest.mat...
```

To verify the MAT-file was created, at the command prompt, type:

```
whos -file mattest.mat
  Name              Size          Bytes  Class

  GlobalDouble      3x3              72  double array (global)
  LocalDouble       3x3              72  double array
  LocalString       1x43             86  char array
```

```
Grand total is 61 elements using 230 bytes
```

Creating a MAT-File in C++

There is a C++ version of matcreat.c in the
matlabroot\extern\examples\eng_mat directory. To see matcreat.cpp,
open the file in the MATLAB Editor.

Reading a MAT-File in C

This program, matdgns.c, illustrates how to use the library routines to read
and diagnose a MAT-file. To see the code, you can open the file in MATLAB
Editor.

After compiling and linking this program, you can view its results:

```
matdgns mattest.mat
Reading file mattest.mat...

Directory of mattest.mat:
GlobalDouble
LocalString
LocalDouble

Examining the header for each variable:
According to its header, array GlobalDouble has 2 dimensions
  and was a global variable when saved
According to its header, array LocalString has 2 dimensions
  and was a local variable when saved
According to its header, array LocalDouble has 2 dimensions
  and was a local variable when saved

Reading in the actual array contents:
According to its contents, array GlobalDouble has 2 dimensions
  and was a global variable when saved
According to its contents, array LocalString has 2 dimensions
  and was a local variable when saved
According to its contents, array LocalDouble has 2 dimensions
  and was a local variable when saved
Done
```

Creating a MAT-File in Fortran

This program, `matdemo1.F`, creates the MAT-file, `matdemo.mat`. To see the code, you can open the file in MATLAB Editor.

Once you have compiled and linked your MAT-file program, you can run the stand alone application you have just produced. This program creates a MAT-file, `matdemo.mat`, that can be loaded into MATLAB. To run the application, depending on your platform, either double-click its icon or enter `matdemo1` at the system prompt:

```
matdemo1
Creating MAT-file matdemo.mat ...
Done creating MAT-file
```

To verify that the MAT-file has been created, at the command prompt, enter:

```
 whos -file matdemo.mat
 Name          Size          Bytes  Class

 Numeric       3x3              72  double array
 String        1x33            66  char array

 Grand total is 42 elements using 138 bytes
```

Note For an example of a Microsoft Windows stand alone program (not MAT-file specific), see `engwindemo.c` in the *matlabroot*\extern\examples\eng_mat directory.

Reading a MAT-File in Fortran

This program, `matdemo2.F`, illustrates how to use the library routines to read the MAT-file created by `matdemo1.F` and describe its contents. To see the code, you can open the file in the MATLAB Editor.

After compiling and linking this program, you can view its results:

```
matdemo2
Directory of Mat-file:
String
```

```
Numeric
Getting full array contents:
  1
Retrieved String
  With size   1-by-  33
  3
Retrieved Numeric
  With size   3-by-  3
```

Compiling and Linking MAT-File Programs

In this section...

Compiling and Linking on UNIX Operating Systems

At run-time, you must tell the UNIX[2] operating system where the API shared libraries reside. These sections provide the necessary UNIX commands, depending on your shell and system architecture.

Setting Run-Time Library Path

Set the library path as follows for the C and Bourne shells. Replace the terms *envvar* and *pathspec* with the appropriate values from the table below.

In the C shell, set the library path using the command:

```
setenv envvar pathspec
```

In the Bourne shell, use:

```
envvar = pathspec:envvar
export envvar
```

Operating System	*envvar*	*pathspec*
Linux®[3]	LD_LIBRARY_PATH	*matlabroot*/bin/glnx86: *matlabroot*/sys/os/glnx86

---

2. UNIX is a registered trademark of The Open Group in the United States and other countries.

3. Linux is a registered trademark of Linus Torvalds.

Operating System	*envvar*	*pathspec*
64-bit Linux	LD_LIBRARY_PATH	*matlabroot*/bin/glnxa64: *matlabroot*/sys/os/glnxa64
64-bit SunSolaris SPARC®	LD_LIBRARY_PATH	*matlabroot*/bin/sol64: *matlabroot*/sys/os/sol64
Apple® Macintosh® (Intel®)	DYLD_LIBRARY_PATH	*matlabroot*/bin/maci: *matlabroot*/sys/os/maci

For example, for the C shell on a Solaris system use:

```
setenv LD_LIBRARY_PATH
matlabroot/bin/sol64:matlabroot/sys/os/sol64
```

and for the Bourne shell:

```
LD_LIBRARY_PATH=matlabroot/bin/sol64:matlabroot/sys/os/sol64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

You can place these commands in a startup script such as ~/.cshrc for C shell or ~/.profile for Bourne shell.

Using the Options File

The MATLAB options file, matopts.sh, lets you use the mex script to easily compile and link MAT-file applications. For example, to compile and link the matcreat.c example, first copy the file using the command:

```
matlabroot/extern/examples/eng_mat/matcreat.c
```

to a writable directory, then use the following command to build it:

```
mex -f matlabroot/bin/matopts.sh matcreat.c
```

If you need to modify the options file for your particular compiler or platform, use the -v switch to view the current compiler and linker settings and then make the appropriate changes in a local copy of the matopts.sh file.

Compiling and Linking on Windows Operating Systems

To compile and link Fortran or C MAT-file programs, use the `mex` script with a MAT options file. The MAT options files reside in *matlabroot*`\bin\win32\mexopts` or *matlabroot*`\bin\win64\mexopts` and are named `*engmatopts.bat`, where `*` represents the compiler type and version.

For example, to compile and link the stand alone MAT application `matcreat.c` using the Microsoft® Visual C++® Version 9.0 compiler on a Microsoft Windows operating system, first copy the file:

    matlabroot\extern\examples\eng_mat\matcreat.c

to a writable directory, and then use the following command to build it:

    mex -f matlabroot\bin\win32\mexopts\msvc90engmatopts.bat matcreat.c

If you need to modify the options file for your particular compiler, use the `-v` switch to view the current compiler and linker settings and then make the appropriate changes in a local copy of the options file.

Required Files from Third-Party Sources

MATLAB requires the following data and library files for building any MAT-file application. You must also distribute these files along with any MAT-file application that you deploy to another system.

Third-Party Data Files

When building a MAT-file application on your system or deploying a MAT-file application to some other system, make sure that the appropriate Unicode data file is installed in the *matlabroot*`/bin/$ARCH` directory. MATLAB uses this file to support Unicode encoding.

For systems that order bytes in a big-endian manner, use `icudt32b.dat`.

For systems that order bytes in a little-endian manner, use `icudt32l.dat`.

Third-Party Libraries

When building a MAT-file application on your system or deploying a MAT-file application to some other system, make sure to install the appropriate libraries in the *matlabroot*/bin/$ARCH directory:

Library File Names by Operating System

Windows	UNIX	Macintosh (Intel)
libmat.dll	libmat.so	libmat.dylib
libmx.dll	libmx.so	libmx.dylib

In addition to these libraries, you must also have all third-party library files that libmat depends on. MATLAB uses these additional libraries to support Unicode character encoding and data compression in MAT-files. These library files must reside in the same directory as libmx.

You can determine what most of these libraries are using the platform-specific methods described below.

Linux or Solaris Operating System

Type the following command:

```
ldd -d libmat.so
```

Macintosh Operating System

Type the following command:

```
otool -L libmat.dylib
```

Windows Operating System

On Windows systems, the third-party product Dependency Walker can be used to diagnose errors related to loading and executing modules. Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module and builds a hierarchical tree diagram of all dependent modules. For each module found, it lists all the functions that are exported by that module, and which of those functions are actually being called by other modules. You can download the Dependency Walker utility from the following Web site:

```
http://www.dependencywalker.com/
```

See the Technical Support solution 1-2RQL4L for information on using the Dependency Walker:

```
http://www.mathworks.com/support/solutions/data/1-2RQL4L.html
```

Drag and drop the file *matlabroot*/bin/win32/libmat.dll or *matlabroot*/bin/win64/libmat.dll into Depends window.

Working Directly with Unicode Encoding

If you need to manipulate Unicode text directly in your application, the latest version of International Components for Unicode (ICU) is freely available online from the IBM® Corporation Web site at:

```
http://icu.sourceforge.net/download
```

2

MATLAB Interface to Shared Libraries

Calling Functions in Shared Libraries

In this section...

What Is a Shared Library?

A shared library is a collection of functions designed to be dynamically loaded by an application at run time. This MATLAB interface supports libraries containing functions programmed in any language, provided the functions have a C interface. MATLAB supports dynamic linking on all supported platforms.

Platform	**Shared Library**	**File Extension**
Microsoft Windows	dynamic link library file	`.dll`
UNIX[4] and Linux[5]	shared object file	`.so`
Apple Macintosh	dynamic shared library	`.dylib`

A shared library needs a *header file*, which provides *signatures* for the functions in the library. A signature, or function prototype, establishes the name of the function and the number and types of its parameters. You need to know the full path of the shared library and its header file.

MATLAB accesses C routines built into external, shared libraries through a command-line interface. This interface lets you load an external library into MATLAB memory and access functions in the library. Although types differ

---

4. UNIX is a registered trademark of The Open Group in the United States and other countries.

5. Linux is a registered trademark of Linus Torvalds.

between the two language environments, in most cases you can pass types to the C functions without converting. MATLAB does this for you.

Details about using a shared library are in the topics:

- "Loading the Library" on page 2-3
- "Viewing Library Functions" on page 2-4
- "Invoking Library Functions" on page 2-7

To call a library function, you need to determine the data passed to and from the function. For information about data, see:

- "Passing Arguments to Shared Library Functions" on page 2-8
- "Manually Converting Data Passed to Functions" on page 2-16
- "Working with Pointers" on page 2-18
- "Working with Structures" on page 2-28

When you are finished working with the shared library, it is important to unload the library to free memory, as described in "Unloading the Library" on page 2-4.

It is possible to call a Microsoft® Visual Basic® Version 6.0 DLL; however, you must create a C header file for the library. A Microsoft® ActiveX® interface might be simpler to use. See "Introducing MATLAB COM Integration" on page 8-2 for more information.

Note The MATLAB Interface to Shared Libraries does not support library functions that have function pointer inputs.

Loading the Library

To give MATLAB software access to functions in a shared library, you must first load the library into memory. After you load the library, you can request information about library functions and call them directly from the MATLAB command line. When you no longer need the library, unload it from memory to conserve memory usage.

To load a shared library into MATLAB, use the `loadlibrary` function. The most common syntax for `loadlibrary` is:

```
loadlibrary('shrlib', 'hfile')
```

where `shrlib` is the shared library file name, and `hfile` is the name of the header file containing the function prototypes. See the `loadlibrary` reference page for variations in the syntax and information on library file extensions.

> **Note** The header file provides signatures for the functions in the library and is a required argument for `loadlibrary`.

For example, you can use `loadlibrary` to load the `libmx` library that defines the MATLAB `mx` routines. The following command creates the full path for the library header file, `matrix.h`:

```
hfile = [matlabroot '\extern\include\matrix.h'];
```

To load the library, type:

```
loadlibrary('libmx', hfile)
```

Unloading the Library

Use the `unloadlibrary` function to unload the library and free up memory. For example:

```
unloadlibrary libmx
```

Viewing Library Functions

- "Viewing Functions in the Command Window" on page 2-4
- "Viewing Functions in a GUI" on page 2-6

Viewing Functions in the Command Window

Use the `libfunctions` command to display information about a library's functions in the MATLAB Command Window. For example, to see what functions are available in the `libmx` library, type:

```
if not(libisloaded('libmx'))
  hfile = [matlabroot '\extern\include\matrix.h'];
  loadlibrary('libmx', hfile);
end
libfunctions libmx
```

MATLAB displays (in part):

```
Functions in library libmx:

mxAddField                      mxGetScalar
mxArrayToString                 mxGetString_730
mxCalcSingleSubscript_730       mxGetUserBits
mxCalloc                        mxIsCell
mxCreateCellArray_730           mxIsChar
mxCreateCellMatrix_730          mxIsClass
        .                               .
        .                               .
        .                               .
```

To view function signatures, use the -full switch. This shows the MATLAB syntax for calling functions written in C. The types used in the argument lists and return values are MATLAB types, not C types. For more information on types, see "C and MATLAB Equivalent Types" on page 2-8. For example, at the command line enter:

```
libfunctions libmx -full
```

MATLAB displays (in part):

```
Functions in library libmx:

[int32, MATLAB array, cstring] mxAddField(MATLAB array, cstring)
[cstring, MATLAB array] mxArrayToString(MATLAB array)
lib.pointer mxCalloc(uint32, uint32)
[MATLAB array, uint32Ptr] mxCreateCellArray_730(uint32, uint32Ptr)
MATLAB array mxCreateCellMatrix_730(uint32, uint32)
[MATLAB array, uint32Ptr] mxCreateCharArray_730(uint32, uint32Ptr)
          .
          .
          .
```

Viewing Functions in a GUI
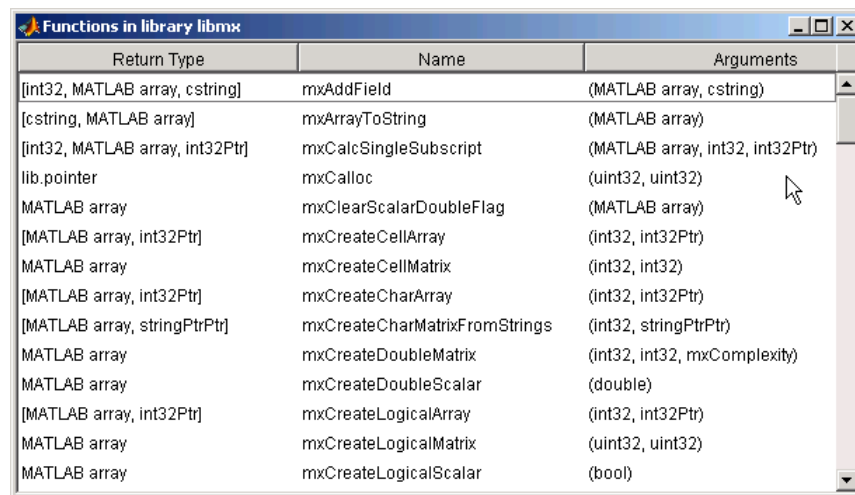
Use the `libfunctionsview` function to get information about functions in a library. MATLAB creates a new window to display the following information:

Heading	Description
Return Type	Types the method returns
Name	Function name
Arguments	Valid types for input arguments

To see the functions in the `libmx` library, type:

```
if not(libisloaded('libmx'))
  hfile = [matlabroot '\extern\include\matrix.h'];
  loadlibrary('libmx', hfile);
end
libfunctionsview libmx
```

MATLAB displays the following window:

The types used in the argument lists and return values are MATLAB types, not C types. For more information on types, see "C and MATLAB Equivalent Types" on page 2-8.

Invoking Library Functions

After loading a shared library into the MATLAB workspace, use the `calllib` function to call functions in the library. The syntax for `calllib` is:

```
calllib('libname', 'funcname', arg1, ..., argN)
```

You need to specify the library name, function name, and any arguments that get passed to the function.

The following example calls functions from the `libmx` library. To load the library, type:

```
if not(libisloaded('libmx'))
  hfile = [matlabroot '\extern\include\matrix.h'];
  loadlibrary('libmx', hfile);
end
```

To create an array `y`, type:

```
y = rand(4, 7, 2);
```

To get information about `y`, type:

```
calllib('libmx', 'mxGetNumberOfElements', y)
```

MATLAB displays the number of elements in the array:

```
ans =
    56
```

Type:

```
calllib('libmx', 'mxGetClassID', y)
```

MATLAB displays the class of the array:

```
ans =
    mxDOUBLE_CLASS
```

For information on how to define the argument types, see "Passing Arguments to Shared Library Functions" on page 2-8.

Passing Arguments to Shared Library Functions

C and MATLAB Equivalent Types

The shared library interface supports all standard scalar C types. The following table shows these C types with their equivalent MATLAB types. MATLAB uses the type from the right column for arguments having the C type shown in the left column. For examples using these arguments, see "Passing Primitive Types" on page 2-12 and "Passing Strings" on page 2-13.

MATLAB Primitive Types

C Type	Equivalent MATLAB Type
char, byte	int8
unsigned char, byte	uint8
short	int16
unsigned short	uint16
int	int32
long (32-bit)	int32
long (64-bit)	int64
unsigned int, unsigned long	uint32
float	single

MATLAB Primitive Types (Continued)

C Type	Equivalent MATLAB Type
double	double
char *	cstring (1xn char array)
*char[]	cell array of strings

The following table shows *extended* MATLAB types in the right column. These are instances of the MATLAB lib.pointer class rather than standard MATLAB types. For information on the lib.pointer class, see "Working with Pointers" on page 2-18. For an example using pointer arguments, see "Passing a Reference" on page 2-13.

MATLAB Extended Types

C Type	Equivalent MATLAB Type
integer pointer types (int *)	(u)int(*size*)Ptr
Null-terminated string passed by value	cstring
Null-terminated string passed by reference (from a libpointer only)	stringPtr
Array of pointers to strings (or one **char)	stringPtrPtr
Matrix of signed bytes	int8Ptr
float *	singlePtr
double *	doublePtr
mxArray *	MATLAB array
void *	voidPtr
void **	voidPtrPtr
type **	Same as *type*Ptr with an added Ptr (e.g., double ** is doublePtrPtr)

Passing Arguments

Here are some important things to note about the input and output arguments shown in the `Functions in library shrlibsample` listing:

- Many arguments (like `int32` and `double`) are similar to their C counterparts. In these cases, you need only to pass in the MATLAB types shown for these arguments.

- Some C arguments (for example, `**double`, or predefined structures), are different from standard MATLAB types. In these cases, you can either pass a standard MATLAB type and let MATLAB convert it for you, or you convert the data yourself using the MATLAB functions `libstruct` and `libpointer`. For more information, see "Manually Converting Data Passed to Functions" on page 2-16.

- C input arguments are often passed by reference. Although MATLAB does not support passing by reference, you can create MATLAB arguments that are compatible with C references. In the `Functions in library shrlibsample` listing, these are the arguments with names ending in `Ptr` and `PtrPtr`. For information on using these types, see "Working with Pointers" on page 2-18.

- C functions often return data in input arguments passed by reference. MATLAB creates additional output arguments to return these values. Note that in the listing in the previous section, all input arguments ending in `Ptr` or `PtrPtr` are also listed as outputs.

Guidelines for Passing Arguments

- Nonscalar arguments must be declared as passed by reference in the library functions.

- If the library function uses single subscript indexing to reference a two-dimensional matrix, keep in mind that C programs process matrices row by row while MATLAB processes matrices by column. To get C behavior from the function, transpose the input matrix before calling the function, and then transpose the function output.

- When passing an array having more than two dimensions, the shape of the array might be altered by MATLAB. To ensure that the array retains its shape, store the size of the array before making the call, and then use

this same size to reshape the output array to the correct dimensions. For example:

```
vs = size(vin)                % Store the original dimensions
vs =
    2    5    2

vout = calllib('shrlibsample','multDoubleArray', vin, 20);

size(vout)                    % Dimensions have been altered
ans =
    2    10

vout = reshape(vout, vs);   % Restore the array to 2-by-5-by-2

size(vout)
ans =
    2    5    2
```

- Use an empty array, [], to pass a NULL parameter to a library function that supports optional input arguments. This is valid only when the argument is declared as a Ptr or PtrPtr as shown by libfunctions or libfunctionsview.

Examples of Passing Data to Shared Libraries

- "Sample Shared Library shrlibsample" on page 2-11
- "Passing Primitive Types" on page 2-12
- "Passing a Reference" on page 2-13
- "Passing Strings" on page 2-13
- "Passing Enumerated Types" on page 2-15

Sample Shared Library shrlibsample

MATLAB software includes a sample external library called shrlibsample. The library is in the directory *matlabroot*\extern\examples\shrlib.

To use the `shrlibsample` library, you first need to either add this directory to your MATLAB path with the command:

```
addpath([matlabroot '\extern\examples\shrlib'])
```

or make the directory your current working directory with the command:

```
cd([matlabroot '\extern\examples\shrlib'])
```

The following example loads the `shrlibsample` library and displays the MATLAB syntax for calling functions in the library:

```
loadlibrary shrlibsample shrlibsample.h
libfunctions shrlibsample -full
```

MATLAB displays:

```
Functions in library shrlibsample:

[double, doublePtr] addDoubleRef(double, doublePtr, double)
double addMixedTypes(int16, int32, double)
[double, c_structPtr] addStructByRef(c_structPtr)
double addStructFields(c_struct)
c_structPtrPtr allocateStruct(c_structPtrPtr)
voidPtr deallocateStruct(voidPtr)
doublePtr multDoubleArray(doublePtr, int32)
[lib.pointer, doublePtr] multDoubleRef(doublePtr)
int16Ptr multiplyShort(int16Ptr, int32)
cstring readEnum(Enum1)
[cstring, cstring] stringToUpper(cstring)
```

Passing Primitive Types

For primitive types, MATLAB automatically converts any argument to the type expected by the external function. For example, you can pass a `double` to a function that expects to receive a `byte` (8-bit integer) and MATLAB does the conversion for you.

The following C function takes arguments that are of types `short`, `int`, and `double`:

```
double addMixedTypes(short x, int y, double z) {
```

```
    return (x + y + z);
}
```

You can pass all of the arguments as type `double` from MATLAB. MATLAB determines what type of data is expected for each argument and performs the appropriate conversions. For example, type:

```
calllib('shrlibsample', 'addMixedTypes', 127, 33000, pi)
```

MATLAB displays:

```
ans =
   3.3130e+004
```

Passing a Reference

MATLAB automatically converts an argument passed by value into an argument passed by reference when the external function prototype defines the argument as a reference. For example, a MATLAB `double` argument passed to a function that expects `double *` is converted to a `double` reference by MATLAB.

`addDoubleRef` is a C function that takes an argument of type `double *`:

```
double addDoubleRef(double x, double *y, double z) {
    return (x + *y + z);
}
```

Call the function with three arguments of type `double`, and MATLAB handles the conversion:

```
calllib('shrlibsample', 'addDoubleRef', 1.78, 5.42, 13.3)
```

MATLAB displays:

```
ans =
   20.5000
```

Passing Strings

For arguments that require `char *`, you can pass a MATLAB string (a character array).

For example, the following C function takes a `char *` input argument:

```
char* stringToUpper(char *input) {
   char *p = input;

   if (p != NULL)
      while (*p!=O)
         *p++ = toupper(*p);
   return input;
}
```

`libfunctions` shows that you can use a MATLAB `cstring` for this input. Type:

```
libfunctions shrlibsample -full
```

Look for the following `stringToUpper` signature:

```
[cstring, cstring] stringToUpper(cstring)
```

Create a MATLAB character array, `str`, and pass it as the input argument:

```
str = 'This was a Mixed Case string';
calllib('shrlibsample', 'stringToUpper', str)
```

MATLAB displays:

```
ans =
   THIS WAS A MIXED CASE STRING
```

Although the input argument that MATLAB passes to `stringToUpper` resembles a reference to type `char`, it is not a true reference data type because it does not contain the address of the MATLAB character array, `str`. When the function executes, it returns the correct result, but does not modify the value in `str`. If you examine `str`, you find it is unchanged. Type:

```
str
```

MATLAB displays:

```
str =
  This was a Mixed Case string
```

Passing Enumerated Types

For arguments defined as C enumerated types, you can pass either the enumeration string or its integer equivalent.

The readEnum function from the shrlibsample library returns the enumeration string that matches the argument passed in. Here is the Enum1 definition and the readEnum function in C:

```
enum Enum1 {en1 = 1, en2, en4 = 4} TEnum1;

char* readEnum(TEnum1 val) {
  switch (val) {
    case 1 :return "You chose en1";
    case 2: return "You chose en2";
    case 4: return "You chose en4";
    default : return "enum not defined";
  }
}
```

In MATLAB, you can express an enumerated type as either the enumeration string or its equivalent numeric value. In the previous example, the TEnum1 definition declares enumeration en4 equal to 4. Call readEnum first with a string:

```
calllib('shrlibsample', 'readEnum', 'en4')
```

MATLAB displays:

```
ans =
    You chose en4
```

Now call it with the equivalent numeric argument 4:

```
calllib('shrlibsample', 'readEnum', 4)
```

MATLAB displays:

```
ans =
    You chose en4
```

Passing References

Many functions in external libraries use arguments that are passed by reference. To enable you to interact with these functions, MATLAB passes what is called a *pointer object* to these arguments. This should not be confused with "passing by reference" in the typical sense of the term. See "Working with Pointers" on page 2-18 for more information.

Passing a NULL Pointer

You can create a NULL pointer to pass to library functions in the following ways:

- Pass a 0 as the argument.

- Use the libpointer function:

      p = libpointer; % no arguments

      p = libpointer('string') % string argument

      p = libpointer('stringPtr') % pointer to a string argument

- Use the libstruct function:

      p = libstruct; % no arguments

Manually Converting Data Passed to Functions

Under most conditions, MATLAB software automatically converts data passed to and from external library functions to the type expected by the external function. However, you may choose to convert your argument data manually. Circumstances under which you might find this advantageous are:

- When you pass the same piece of data to a series of library functions, you can convert it once manually before the call to the first function rather than having MATLAB convert it automatically on every call. This reduces the number of unnecessary copy and conversion operations.

- When you pass large structures, you can save memory by creating MATLAB structures that match the shape of the C structures used in the external function instead of using generic MATLAB structures. The libstruct function creates a MATLAB structure modeled from a C

structure taken from the library. See "Working with Structures" on page 2-28 for more information.

- When an argument to an external function uses more than one level of referencing (e.g., `double **`), you must pass a reference created using the `libpointer` function rather than relying on MATLAB to convert the type automatically.

Using C++ Libraries

To load C++ libraries, you must define the function prototypes as `extern "C"` in the library header file. For example, the following function prototype from the file `mex.h` shows the syntax to use for each function:

```
#ifdef __cplusplus
extern "C" {
#endif
void mexFunction(
    int          nlhs,
    mxArray      *plhs[],
    int          nrhs,
    const mxArray *prhs[]
);
#ifdef __cplusplus
}
#endif
```

Another approach to using C++ libraries is to generate a prototype M-file that contain aliases for the mangled C++ function names. Use the original (premangled) function names as the aliases for the C++ functions. Generate the M-file with the `mfilename` option of the `loadlibrary` function and then determine which functions in the library you want to make available by defining aliases for these functions.

Working with Pointers

You can pass most arguments to an external function by value, even when the prototype for that function declares the argument to be a reference. The calllib function uses the header file to determine how to pass the function arguments.

There are times, however, when it is useful to pass MATLAB arguments that are the equivalent of C references:

- You want to modify the data in the input arguments.

- You are passing large amounts of data, and you don't want MATLAB to make copies of the data.

- The library stores and uses the pointer for a period of time so you want the M-code to control the lifetime of the pointer object.

In these cases, you can use the libpointer function to construct a pointer object of a specified type (for structures use libstruct).

Constructing a Reference with the libpointer Function

To construct a reference, use the function libpointer with this syntax:

```
p = libpointer('type', 'value')
```

For example, create a pointer `pv` to an `int16` value. The `type` of pointer you are creating is always the data type (`int16`, in this case) suffixed by the letters `Ptr`:

```
v = int16(485);
pv = libpointer('int16Ptr', v);
```

The value `pv` is an instance of a MATLAB `lib.pointer` class. The `lib.pointer` class has the properties `Value` and `DataType`. You can read and modify these properties with the `get` and `set` functions:

```
get(pv)
```

MATLAB displays:

```
     Value: 485
  DataType: 'int16Ptr'
```

The `lib.pointer` class method `setdatatype` is described in "Obtaining the Function's Return Values" on page 2-20:

```
methods(pv)
```

MATLAB displays:

```
Methods for class lib.pointer:
   disp        plus         setdatatype
   isNull      reshape
```

Creating a Reference to a Primitive Type

The following example illustrates how to construct and pass a pointer to type `double`, and how to interpret the output data. The function `multDoubleRef` takes one input that is a reference to a `double` and returns the same.

Here is the C function:

```
double *multDoubleRef(double *x) {
    *x *= 5;
    return x;
}
```

Construct a reference, xp, to input data, x, and verify its contents:

```
x = 15;
xp = libpointer('doublePtr', x);
get(xp)
```

MATLAB displays:

```
     Value: 15
  DataType: 'doublePtr'
```

Now call the function and check the results:

```
calllib('shrlibsample', 'multDoubleRef', xp);
get(xp, 'Value')
```

MATLAB displays:

```
ans =
    75
```

Reference xp is not a true pointer as it would be in a language like C. That is, even though it was constructed as a reference to x, it does not contain the address of x. So, when the function executes, it modifies the Value property of xp, but it does not modify the value in x. If you now examine x, you find that its original value is unchanged:

```
x

x =
    15
```

Obtaining the Function's Return Values

In the previous example, the result of the function called from MATLAB could be obtained by examining the modified input reference. But this function also returns data in its output arguments that may be useful.

The MATLAB prototype for this function (returned by libfunctions -full) indicates that MATLAB returns two outputs. The first is an object of class lib.pointer; the second is the Value property of the doublePtr input argument:

```
libfunctions shrlibsample -full
    [lib.pointer, doublePtr] multDoubleRef(doublePtr)
```

Run the example once more, but this time check the output values returned:

```
x = 15;
xp = libpointer('doublePtr', x);
[xobj, xval] = calllib('shrlibsample', 'multDoubleRef', xp)
```

MATLAB displays:

```
xobj =
    lib.pointer
xval =
      75
```

Like the input reference argument, the first output, xobj, is an object of class lib.pointer. You can examine this output, but first you need to initialize its data type and size as these factors are undefined when returned by the function. Use the setdatatype method defined by class lib.pointer to set the data type to doublePtr and the size to 1-by-1. Once initialized, you can examine outputs.

The first output is xobj:

```
setdatatype(xobj, 'doublePtr', 1, 1)
get(xobj)
```

MATLAB displays:

```
ans =
        Value: 75
     DataType: 'doublePtr'
```

The second output, xval, is a double copied from the Value of input xp.

Creating a Reference by Offsetting from an Existing libpointer

You can use the plus operator (+) to create a new pointer that is offset from an existing pointer by a scalar numeric value. This operation applies only to pointers of numeric data types. For example, suppose you create a libpointer to the vector x:

```
x = 1:10;
xp = libpointer('doublePtr',x);
xp.Value
```

MATLAB displays:

```
ans =
     1     2     3     4     5     6     7     8     9    10
```

You can now use the plus operator to create a new libpointer that is offset from the xp:

```
xp2 = xp+4;
xp2.Value
```

MATLAB displays:

```
ans =
     5     6     7     8     9    10
```

Note that the new pointer (xp2 in this example) is valid only as long as the original pointer exists.

Creating a Structure Reference

Creating a reference argument to a structure is not much different than using a reference to a primitive type. The function shown here takes a reference to a structure of type c_struct as its only input. It returns an output argument that is the sum of all fields in the structure. It also modifies the fields of the input argument:

```
double addStructByRef(struct c_struct *st) {
    double t = st->p1 + st->p2 + st->p3;
    st->p1 = 5.5;
    st->p2 = 1234;
    st->p3 = 12345678;
    return t;
 }
```

Passing the Structure Itself

Although this function expects to receive a structure reference input, it is easier to pass the structure itself and let MATLAB do the conversion to a reference. This example passes a MATLAB structure, sm, to the function addStructByRef. MATLAB returns the correct value in the output, x, but does not modify the contents of the input, sm, since sm is not a reference:

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
x = calllib('shrlibsample', 'addStructByRef', sm)
```

MATLAB displays:

```
x =
        1177
```

Passing a Structure Reference

The second part of this example passes the structure by reference. This time, the function receives a pointer to the structure and is then able to modify the structure fields.

```
sp = libpointer('c_struct', sm);
calllib('shrlibsample', 'addStructByRef', sp)
```

MATLAB displays:

```
ans =
        1177
```

Type:

```
get(sp, 'Value')
```

MATLAB displays:

```
ans =
    p1: 5.5000
    p2: 1234
    p3: 12345678
```

Passing a Pointer to the First Element of an Array

In cases where a function defines an input argument that is a pointer to the first element of a data array, the `calllib` function automatically passes an argument that is a pointer of the correct type to the first element of data in the MATLAB vector or matrix. For example, the following C function `sum` requires an argument that is a pointer to the first element of an array of shorts (`int16`).

Suppose you define the following MATLAB variables:

```
Data = 1:100;
lp = libpointer('int16Ptr',Data);
shortData = int16(Data);
```

The signature of the C function `sum` in library `libname` is:

```
int sum(int size, short* data);
```

The following statements give the same answer:

```
summed_data = calllib('libname','sum',100,Data);
summed_data = calllib('libname','sum',100,shortData);
summed_data = calllib('libname','sum',100,lp);
```

The `size` and `data` arguments must match. That is, length of the data vector must be equal to the specified size. For example:

```
% sum last 50 elements
summed_data = calllib('libname','sum',50,Data(50:100));
```

Creating a Void Pointer to a String

Suppose you want to create a `voidPtr` that points to a string as an input argument. In C, characters are represented as unsigned eight-bit integers. Therefore, you must first cast the string to this MATLAB type before creating a variable of type `voidPtr`.

You can create a variable of the correct type and value using `libpointer` as follows:

```
str = 'string variable';
vp = libpointer('voidPtr',[uint8(str) 0]);
```

To obtain the character string from the pointer, type:

```
char(vp.Value)
```

MATLAB displays:

```
ans =
string variable
```

To confirm the pointer type, type:

```
vp.DataType
```

MATLAB displays:

```
ans =
voidPtr
```

You can call a function that takes a `voidPtr` to a string as an input argument using the following syntax because MATLAB automatically converts an argument passed by value into an argument passed by reference when the external function prototype defines the argument as a reference:

```
func_name(uint8(str))
```

Note that while MATLAB converts the argument from a value to a reference, it must be of the correct type.

Memory Allocation for an External Library

In general, a valid memory address is passed each time you pass a MATLAB variable to a library function. You need to explicitly allocate memory only if the library provides a memory allocation function that you are required to use.

When to Use libpointer

You should use a `libpointer` object in cases where the library is going to store the pointer and access the buffer over a period of time. In these cases, you need to ensure that MATLAB has control over the lifetime of the buffer and to prevent copies of the data from being made. The following pseudo code is an example of asynchronous data acquisition that shows how to use `libpointer` in this type of situation.

Suppose an external library has the following functions:

```
AcquireData(int points, short *buffer)
IsAquistionDone(void)
```

First, create a pointer to a buffer of 1024 points:

```
BufferSize = 1024;
pBuffer = libpointer('int16Ptr',1:BufferSize);
```

Then, begin acquiring data and wait in a loop until it is done:

```
calllib('lib_name','AcquireData,BufferSize,pbuffer);
while (~calllib('lib_name','IsAcquisitionDone')
 pause(0.1)
end
```

The following statement reads the data in the buffer:

```
result = pBuffer.Value;
```

When the library is done with the buffer, clear the MATLAB variable:

```
clear pBuffer
```

Reference Pointers

Arguments that have more than one level of referencing (e.g., uint16 **) are referred to here as reference pointers. In MATLAB, these argument types are named with the suffix PtrPtr (for example, uint16PtrPtr). See the output of libfunctionsview or libfunctions -full for examples of this type.

When calling a function that takes a reference pointer argument, you can use a reference argument instead and MATLAB converts it to the reference pointer. For example, the external allocateStruct function expects a c_structPtrPtr argument. Type:

```
libfunctions shrlibsample -full
```

Look for the following allocateStruct signature:

```
c_structPtrPtr allocateStruct(c_structPtrPtr)
```

Here is the C function:

```
void allocateStruct(struct c_struct **val) {
    *val=(struct c_struct*) malloc(sizeof(struct c_struct));
    (*val)->p1 = 12.4;
    (*val)->p2 = 222;
    (*val)->p3 = 333333;
}
```

Although the prototype says that a c_structPtrPtr is required, you can use a c_structPtr and let MATLAB do the second level of conversion. Create a reference to an empty structure argument and pass it to allocateStruct:

```
sp = libpointer('c_structPtr');
calllib('shrlibsample', 'allocateStruct', sp)
get(sp)
```

MATLAB displays:

```
ans =
         Value: [1x1 struct]
      DataType: 'c_structPtr'
```

Type:

```
get(sp, 'Value')
```

MATLAB displays:

```
ans =
    p1: 12.4000
    p2: 222
    p3: 333333
```

Free memory using the command:

```
calllib('shrlibsample', 'deallocateStruct', sp)
```

Working with Structures

In this section...
"Structure Argument Requirements" on page 2-28
"Working with Structures Examples" on page 2-28
"Finding Structure Field Names" on page 2-29
"Example of Passing a MATLAB Structure" on page 2-30
"Passing a libstruct Object" on page 2-31
"Using the Structure as an Object" on page 2-34

Structure Argument Requirements

For library functions that take structure arguments, you need to pass structures that have field names that are the same as those in the structure definitions in the library.

The following guidelines apply to MATLAB structures passed to external library functions:

- If a structures contains fewer fields than defined in the library structure, MATLAB sets undefined and uninitialized fields to zero.

- Every structure field name you use must match a field name in the structure definition. Structure names are case sensitive.

- Structures cannot contain fields that are not in the library structure definition.

When you create and initialize the structure, you do not necessarily have to match the data types of numeric fields. MATLAB converts to the correct numeric type for you when you make the call using the `calllib` function.

Working with Structures Examples

[instructions for opening shrlibsample code]

Examples in this topic are:

- "Example of Finding Structure Field Names" on page 2-29
- "Example of Passing a MATLAB Structure" on page 2-30
- "Example of Passing a libstruct Object" on page 2-33
- "Using the Structure as an Object" on page 2-34

Finding Structure Field Names

To determine the names and data types of structure fields, you can:

- Consult the library documentation.
- Look at the structure definition in the header file you used to load the library into MATLAB.
- Use the `libstruct`, as described in "Example of Finding Structure Field Names" on page 2-29.

Example of Finding Structure Field Names

You can also determine the field names of an externally defined structure from MATLAB using the following procedure.

1 Use `libfunctionsview` to display the signatures for all functions in the library. `libfunctionsview` shows the names of the structures used by each function. For example, when you type:

```
libfunctionsview shrlibsample
```

MATLAB opens a window displaying function signatures for the `shrlibsample` library. The line showing the `addStructFields` function reads:

```
double addStructFields (c_struct)
```

2 If the function you are using takes a structure argument, get the structure type from the `libfunctionsview` display, and invoke the `libstruct` function on that type. `libstruct` returns an object that is modeled on the structure as defined in the library:

```
s = libstruct('c_struct');
```

2-29

3 To get the names of the structure fields from the object returned by `libstruct`, type:

```
get(s)
```

MATLAB displays:

```
p1: 0
p2: 0
p3: 0
```

4 Initialize the fields to the values you want to pass to the library function and make the call using `calllib`:

```
s.p1 = 476;    s.p2 = -299;    s.p3 = 1000;
calllib('shrlibsample','addStructFields',s);
```

Example of Passing a MATLAB Structure

As with other data types, when an external function takes a structure argument (such as a C structure), you can pass a MATLAB structure to the function in its place.

The sample shared library, `shrlibsample`, defines the following C structure and function:

```
struct c_struct {
    double p1;
    short p2;
    long  p3;
};

double addStructFields(struct c_struct st) {
    double t = st.p1 + st.p2 + st.p3;
    return t;
}
```

The following example passes a MATLAB structure to `addStructFields`. To load the library, type:

```
if not(libisloaded('shrlibsample'))
  addpath([matlabroot '\extern\examples\shrlib']);
```

```
    loadlibrary shrlibsample shrlibsample.h;
  end
```

Create the structure, `sm`, with three `double` fields. MATLAB converts the fields to the `double`, `short`, and `long` data types defined in the C structure, `c_struct`. For example, type

```
  sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
  calllib('shrlibsample', 'addStructFields', sm)
```

MATLAB displays:

```
  ans =
        1177
```

Passing a libstruct Object

- "Preconverting a MATLAB Struct with libstruct" on page 2-32

- "Creating an Empty libstruct Object" on page 2-32

- "libstruct Requirements for Structures" on page 2-33

- "Example of Passing a libstruct Object" on page 2-33

When you pass a structure to an external function, MATLAB makes sure that the structure being passed matches the library's definition for that structure type. It must contain all the necessary fields defined for that type and each field must be of the expected data type. For any fields that are missing in the structure being passed, MATLAB creates an empty field of that name and initializes its value to zero. For any fields that have a data type that doesn't match the structure definition, MATLAB converts the field to the expected type.

When working with small structures, it is efficient enough to have MATLAB do this work for you. You can pass the original MATLAB structure with `calllib` and let MATLAB handle the conversions automatically. However, when working with repeated calls that pass one or more large structures, it may be to your advantage to convert the structure manually before making any calls to external functions. In this way, you save processing time by converting the structure data only once at the start rather than at each

function call. You can also save memory if the fields of the converted structure take up less space than the original MATLAB structure.

Preconverting a MATLAB Struct with libstruct

You can convert a MATLAB structure to a C-style structure derived from a specific type definition in the library in one step. Call the `libstruct` function, passing in the name of the structure type from the library, and the original structure from MATLAB. The syntax for `libstruct` is:

```
s = libstruct('structtype', mlstruct)
```

The value `s` returned by this function is called a *libstruct object*. Although it is a MATLAB object, it behaves like a MATLAB structure. The fields of this new "structure" are derived from the external structure type specified by `structtype` in the syntax above.

For example, to convert a MATLAB structure, `sm`, to a `libstruct` object, `sc`, that is derived from the `c_struct` structure type, type:

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
sc = libstruct('c_struct', sm);
```

All of fields in the original structure `sm` are of type `double`. The object `sc`, returned from the `libstruct` function, has fields that match the `c_struct` structure type. These fields are `double`, `short`, and `long`.

Note You can only use `libstruct` on scalar structures.

Creating an Empty libstruct Object

You can create an empty libstruct object by calling `libstruct` with only the `structtype` argument. For a type specified by `structtype`, the following syntax constructs an object with all the required fields and with each field initialized to zero:

```
s = libstruct('structtype')
```

libstruct Requirements for Structures

When converting a MATLAB structure to a libstruct object, the structure to be converted must adhere to the same guidelines that were documented for MATLAB structures passed directly to external functions. See "Structure Argument Requirements" on page 2-28.

Example of Passing a libstruct Object

Repeat the addStructFields example, this time converting the structure to one of type c_struct before calling the function:

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
sc = libstruct('c_struct', sm);
get(sc)
```

MATLAB displays:

```
    p1: 476
    p2: -299
    p3: 1000
```

Now call the function, passing the structure sc:

```
calllib('shrlibsample', 'addStructFields', sc)
```

MATLAB displays:

```
ans =
    1177
```

Compare this example with "Example of Passing a MATLAB Structure" on page 2-30.

Note When passing manually converted structures, the structure passed must be of the same type as that used by the external function. For example, you cannot pass a structure of type records to a function that expects type c_struct.

Using the Structure as an Object

- "Determining the Size of a lib.c_struct Object" on page 2-34
- "Accessing lib.c_struct Fields" on page 2-34

The value returned by `libstruct` is not a MATLAB structure. It is an instance of a class called `lib.c_struct`, as seen by examining the output of `whos`. Type:

```
whos
```

MATLAB displays (in part):

```
  Name       Size                    Bytes  Class

  sc         1x1                            lib.c_struct
  sm         1x1                      396  struct array
```

Determining the Size of a lib.c_struct Object

You can use the `lib.c_struct` class method `structsize` to obtain the size of a `lib.c_struct` object:

```
sc.structsize
```

MATLAB displays:

```
ans =
   16
```

Accessing lib.c_struct Fields

The fields of this structure are implemented as properties of the `lib.c_struct` class. You can read and modify any of these fields using the MATLAB object-oriented functions, `set` and `get`:

```
sc = libstruct('c_struct');
set(sc, 'p1', 100, 'p2', 150, 'p3', 200);
get(sc)
```

MATLAB displays:

```
p1: 100
p2: 150
p3: 200
```

You can read and modify the fields by treating them like MATLAB structure fields:

```
sc.p1 = 23;
sc.p1
```

MATLAB displays:

```
ans =
    23
```

3

Calling C and Fortran Programs from MATLAB Command Line

Although the MATLAB product is a complete, self-contained environment for programming and manipulating data, it is often useful to interact with data and programs external to the MATLAB environment. MATLAB provides a C and Fortran API for programs written in these languages. You use the API functions in your programs and build them into binary MEX-files, which you call from the MATLAB command line.

Using MEX-Files to Call C and Fortran Programs

In this section...

What Are MEX-Files?

You can call your own C, C++, or Fortran subroutines from the MATLAB command line as if they were built-in functions. These programs are called binary *MEX-files*, which are dynamically linked subroutines that the MATLAB interpreter loads and executes. MEX stands for "MATLAB executable."

Note MATLAB supports MEX-files created in C++, with some limitations. For more information, see "Creating C++ MEX-Files" on page 4-9.

MEX-files have several applications:

• Large pre-existing C and Fortran programs can be called from MATLAB without having to be rewritten as M-files.

• Performance-critical routines can be replaced with handcrafted C implementations.

MEX-files are not appropriate for all applications. MATLAB is a high-productivity environment whose specialty is eliminating time-consuming, low-level programming in compiled languages like Fortran or C. In general, you should do most of your programming in MATLAB. Do not use MEX-files unless your application requires it.

Definition of MEX

The term mex has different meanings, as shown in the following table:

MEX Term	Definition
source MEX-file	C, C++, or Fortran source code file.
binary MEX-file	Dynamically linked subroutine executed in the MATLAB environment.
mex function library	MATLAB C and Fortran API library to perform operations in the MATLAB environment .
mex build script	MATLAB function to create a binary file from a source file.

Overview of a Source MEX-File

This section provides an overview of the elements of a source MEX-file and what you need to get started. To see a C language example, see "Creating a Source MEX-File" on page 3-4. For information about using specific MATLAB C and Fortran API library functions, see "Workflow of a MEX-File" on page 3-9.

Although you can create MEX-files in C, C++ or Fortran, for clarity, this topic is in the context of a C language program. For language-specific instructions for creating MEX-files, see Chapter 4, "Creating C Language MEX-Files" and Chapter 5, "Creating Fortran MEX-Files".

Users who create source MEX-files should have the tools and knowledge to modify a C program. In particular, you need a compiler supported by MATLAB. For an up-to-date list of supported compilers, see Technical Note 1601.

The source code that performs some function you want to use in conjunction with MATLAB software functionality is called a *computational routine*. If you created a stand alone C program for this code, it would have a main() function. MATLAB communicates with your MEX-file using a *gateway routine*. The MATLAB function that creates the gateway routine is *mexfunction*. You use mexfunction instead of main() in your source file.

MATLAB stores arrays in an *mxArray* type. Use mxArray in your C program to pass MATLAB data to and from your MEX-file.

The *MATLAB C and Fortran API* is the function reference for working with mxArray. This API has several libraries. Functions in the mx library create and manipulate MATLAB arrays. These functions are listed in the MX Array Manipulation category. Functions in the mex library perform operations in the MATLAB workspace. These functions are listed in the MEX-Files category.

Overview of Creating a Binary MEX-File

To create a binary MEX-file, you need to:

- Assemble your functions and the MATLAB API functions into one or more C source files.

- Write a gateway function in one of your C source files.

- Use the MATLAB *mex* function, called a build script, to build a binary MEX-file.

- Use your binary MEX-file in MATLAB like any M-file or built-in function.

Configuring Your Environment

Before you start building binary MEX-files, you should select your default compiler and test an existing source MEX-file. For more information about compilers, and for step-by-step instructions for compiling sample programs, see "Building MEX-Files" on page 3-22.

Creating a Source MEX-File

Suppose you have some C code, called arrayProduct, that multiplies an n-dimensional array y by a scalar value x and returns the results in array z. It may look something like the following:

```
void arrayProduct(double x, double *y, double *z, int n)
{
  int i;

  for (i=0; i<n; i++) {
    z[i] = x * y[i];
```

```
    }
  }
```

If `x = 5` and `y` is an array with values 1.5, 2, and 9, then calling:

```
  arrayProduct(x,y,z,n)
```

creates an array `z` with the values 7.5, 10, and 45.

The following steps show you how to call this function in MATLAB using a
MATLAB matrix, by creating the MEX-file `arrayProduct`.

1 "Create Your MEX Source File" on page 3-5

2 "Create a Gateway Routine" on page 3-5

3 "Use Preprocessor Macros" on page 3-6

4 "Verify Input and Output Parameters" on page 3-7

5 "Read Input Data" on page 3-7

6 "Prepare Output Data" on page 3-8

7 "Perform Calculation" on page 3-8

8 "Build the Binary MEX-File" on page 3-8

9 "Test the MEX-File" on page 3-9

Create Your MEX Source File

Open the MATLAB Editor and copy your code into a new file. Save the file on
your MATLAB path, for example, in `c:\work`, and name it `arrayProduct.c`.
This is your computational routine and the name of your MEX-file.

Create a Gateway Routine

At the beginning of the file, add the header file:

```
  #include "mex.h"
```

Add comments:

```
/*
 * arrayProduct.c
 * Multiplies an input scalar times a 1xN matrix
 * and outputs a 1xN matrix
 *
 * This is a MEX-file for MATLAB.
*/
```

After the computational routine, add the gateway routine `mexFunction`:

```
/* The gateway function */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
/* variable declarations here */

/* code here */
}
```

Use Preprocessor Macros

The `mx*` and `mex*` functions use MATLAB preprocessor macros for cross-platform flexibility.

Edit your computational routine to use `mwSize` for `mxArray` size `n` and index `i`.

```
void arrayProduct(double x, double *y, double *z, mwSize n)
{
  mwSize i;

  for (i=0; i<n; i++) {
    z[i] = x * y[i];
  }
}
```

Verify Input and Output Parameters

In this example, there are two input arguments (a matrix and a scalar) and one output argument (the product). To check that the number of input arguments nrhs is two and the number of output arguments nlhs is one, put the following code inside the mexFunction routine:

```
/* check for proper number of arguments */
if(nrhs!=2) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nrhs",
                        "Two inputs required.");
}
if(nlhs!=1) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:nlhs",
                        "One output required.");
}
```

To validate the input values, enter:

```
/* make sure the first input argument is scalar */
if( !mxIsDouble(prhs[0]) ||
     mxIsComplex(prhs[0]) ||
     mxGetNumberOfElements(prhs[0])!=1 ) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notScalar",
                        "Input multiplier must be a scalar.");
}
```

The second input argument must be a row vector.

```
/* check that number of rows in second input argument is 1 */
if(mxGetM(prhs[1])!=1) {
    mexErrMsgIdAndTxt("MyToolbox:arrayProduct:notRowVector",
                        "Input must be a row vector.");
}
```

Read Input Data

Put the following declaration statements at the beginning of your mexFunction:

```
double multiplier;      /* input scalar */
```

```
double *inMatrix;      /* 1xN input matrix */
mwSize ncols;          /* size of matrix */
```

Add these statements to the code section of `mexFunction`:

```
/* get the value of the scalar input  */
multiplier = mxGetScalar(prhs[0]);

/* create a pointer to the real data in the input matrix  */
inMatrix = mxGetPr(prhs[1]);

/* get dimensions of the input matrix */
ncols = mxGetN(prhs[1]);
```

Prepare Output Data

Put the following declaration statement after your input variable declarations:

```
double *outMatrix;     /* output matrix */
```

Add these statements to the code section of `mexFunction`:

```
/* create the output matrix */
plhs[0] = mxCreateDoubleMatrix(1,ncols,mxREAL);

/* get a pointer to the real data in the output matrix */
outMatrix = mxGetPr(plhs[0]);
```

Perform Calculation

The following statement executes your function:

```
/* call the computational routine */
arrayProduct(multiplier,inMatrix,outMatrix,ncols);
```

Build the Binary MEX-File

Your source file should look something like `arrayProduct.c`, located in your *matlabroot*/extern/examples/mex directory. To see the contents of `arrayProduct.c`, open the file in the MATLAB Editor.

To build the binary MEX-file, at the MATLAB command prompt, type:

```
mex arrayProduct.c
```

Test the MEX-File

Type:

```
s = 5;
A = [1.5, 2, 9];
B = arrayProduct(s,A)
```

MATLAB displays:

```
B =
    7.5000   10.0000   45.0000
```

To test error conditions, type:

```
arrayProduct
```

MATLAB displays:

```
??? Error using ==> arrayProduct
Two inputs required.
```

Workflow of a MEX-File

This section discusses MATLAB API functions for handling the basic workflow of a MEX-file and uses C language code snippets for illustration. For an example of a complete C program, see "Creating a Source MEX-File" on page 3-4. Unless otherwise specified, in this section the term "MEX-file" refers to a source file.

Some basic programming tasks are:

- "Creating a Gateway Function" on page 3-10

- "Declaring Data Structures" on page 3-10

- "Managing Input and Output Parameters" on page 3-10

- "Validating Inputs" on page 3-11

- "Allocating and Freeing Memory" on page 3-11

- "Manipulating Data" on page 3-12
- "Displaying Messages to the User" on page 3-13
- "Handling Errors" on page 3-13
- "Cleaning Up and Exiting" on page 3-14

Creating a Gateway Function

Use the `mexfunction` function in your C source file as the interface between your code and MATLAB. Place this function after your computational routine and any other functions in your source.

The signature for `mexfunction` is:

```
void
mexFunction(int nlhs, mxArray *plhs[], ...
            int nrhs, const mxArray *prhs[]);
```

Declaring Data Structures

Use type `mxArray` to handle MATLAB arrays. The following statement declares an `mxArray` named `myData`:

```
mxArray *myData;
```

To define the values of `myData`, use one of the `mxCreate*` functions. Some useful array creation routines are `mxCreateNumericArray`, `mxCreateCellArray`, and `mxCreateCharArray`. For example, the following statement allocates an m-by-1 floating-point `mxArray` initialized to `0`:

```
myData = mxCreateDoubleMatrix(m, 1, mxREAL);
```

C programmers should note that data in a MATLAB array is in column-major order. (For an illustration, see "Data Storage" on page 3-17.) Use the MATLAB `mxGet*` array access routines, described in "Manipulating Data" on page 3-12, to read data from an `mxArray`.

Managing Input and Output Parameters

MATLAB passes data to and from MEX-files in a highly regulated way, described in "Required Parameters" on page 4-3.

Input parameters (found in the `prhs` array) are read only; your MEX-file should not modify them. Changing data in an input parameter may produce undesired side effects.

You also must take care when using an input parameter to create output data or any data used locally in your MEX-file. This is because of the way MATLAB handles MEX-file cleanup after processing. For an example, see the troubleshooting topic "Incorrectly Constructing a Cell or Structure mxArray" on page 3-57.

If you want to copy an input array into your local `myData` array, call `mxDuplicateArray` to make of copy of the input array before using it. For example:

```
mxArray *myData = mxCreateStructMatrix(1,1,nfields,fnames);
mxSetField(myData,0,"myFieldName",mxDuplicateArray(prhs[0]));
```

Validating Inputs

Good programming practice requires you to validate inputs to your function. MATLAB provides `mxIs*` routines for this purpose. The `mxIsClass` function is a general-purpose way to test an `mxArray`.

For example, if your second input argument (identified by `prhs[1]`) must be a full matrix of real numbers, you can use the following statements to check this condition:

```
if(mxIsSparse(prhs[1]) ||
   mxIsComplex(prhs[1]) ||
   mxIsClass(prhs[1],"char")) {
      mexErrMsgTxt("input2 must be full matrix of real values.");
 }
```

This is not an exhaustive check. You may also need to test for structures, cell arrays, function handles, and MATLAB objects.

Allocating and Freeing Memory

Although MATLAB performs cleanup of MEX-file variables, as described in "Automatic Cleanup of Temporary Arrays" on page 4-30, we recommend that

binary MEX-files destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism.

MATLAB provides functions, such as `mxMalloc` and `mxFree`, to manage memory. Use these functions instead of their standard C library counterparts because they let MATLAB manage memory and perform initialization and cleanup.

For information on how MATLAB allocates memory for arrays and data structures, see "Memory Allocation" in the Programming Fundamentals documentation.

You need to allocate memory for variables that your MEX-file uses. If the first input to your function (`prhs[0]`) is a string, in order to manipulate the string, you need to create a buffer `buf` of size `buflen`. The following statements declare these variables:

```
char *buf;
int buflen;
```

The size of the buffer is dependent on the number of dimensions of your input array and the size of the data in the array. This statement calculates the size of `buflen`:

```
buflen = mxGetN(prhs[0])*sizeof(mxChar)+1;
```

Now we can allocate memory for `buf`:

```
buf = mxMalloc(buflen);
```

If `buf` is not returned as a `plhs` output parameter (as described in "Cleaning Up and Exiting" on page 3-14), then you should free its memory as follows:

```
mxFree(buf);
```

Manipulating Data

The `mxGet*` array access routines get references to the data in an `mxArray`. Use these routines to modify data in your MEX-file. Each function provides access to specific information in the `mxArray`. Some useful functions are

mxGetData, mxGetPr, mxGetM, and mxGetString. Many of these functions have corresponding mxSet* routines to allow you to modify values in the array.

The following statements read the input string prhs[0] into a C-style string buf:

```
char *buf;
int buflen;
int status;
buflen = mxGetN(prhs[0])*sizeof(mxChar)+1;
buf = mxMalloc(buflen);
status = mxGetString(prhs[0], buf, buflen);
```

Displaying Messages to the User

Use the mexPrintf function, as you would a C printf function, to print a string in the MATLAB Command Window. Use the mexErrMsgIdAndTxt and mexWarnMsgIdAndTxt functions to print error and warning information in the Command Window.

For example, using the variables declared in the previous example, you can print the input string prhs[0] as follows:

```
if (mxGetString(prhs[0], buf, buflen) == 0) {
    mexPrintf("The input string is:  %s\n", buf);
}
```

Handling Errors

The mexErrMsgIdAndTxt function prints error information and terminates your binary MEX-file. The mexWarnMsgIdAndTxt function prints information, but does not terminate the MEX-file.

```
if (mxIsChar(prhs[0])) {
    if (mxGetString(prhs[0], buf, buflen) == 0) {
        mexPrintf("The input string is:  %s\n", buf);
    }
    else {
        mexErrMsgIdAndTxt("MyProg:ConvertString",
```

```
            "Could not convert string data.");
        // exit MEX-file
    }
}
else {
    mexWarnMsgIdAndTxt("MyProg:InputString",
        "Input should be a string to print properly.");
}

// continue with processing
```

Cleaning Up and Exiting

As described in "Allocating and Freeing Memory" on page 3-11, destroy any temporary arrays and free any dynamically allocated memory, except if such an `mxArray` is returned in the output argument list, returned by `mexGetVariablePtr`, or used to create a structure. Also, never delete input arguments.

Use `mxFree` to free memory allocated by the `mxCalloc`, `mxMalloc`, or `mxRealloc` functions. Use `mxDestroyArray` to free memory allocated by the `mxCreate*` functions.

Using Binary MEX-Files

Binary MEX-files are subroutines produced from C/C++ or Fortran source code. They behave just like M-files and built-in functions. While M-files have a platform-independent extension `.m`, MATLAB identifies MEX-files by platform-specific extensions. The following table lists the platform-specific extensions for MEX-files.

Binary MEX-File Extensions

Platform	Binary MEX-File Extension
Linux[6] (32-bit)	mexglx
Linux x86-64	mexa64

---

6. Linux is a registered trademark of Linus Torvalds.

Binary MEX-File Extensions (Continued)

Platform	Binary MEX-File Extension
Apple Macintosh (Intel)	`mexmaci`
64-bit Sun Solaris SPARC	`mexs64`
Microsoft Windows (32-bit)	`mexw32`
Windows x64	`mexw64`

You call MEX-files exactly as you call any M-function. For example, on a Windows platform, there is a binary MEX-file called `histc.mexw32` in one of the MATLAB toolbox directories (*matlabroot*\toolbox\matlab\datafun) that performs a histogram count. The file `histc.m` contains the help text documentation. When you call `histc` from MATLAB, the dispatcher looks through the list of directories on the MATLAB search path. It scans each directory looking for the first occurrence of a file named `histc` with either the corresponding file name extension from the table or `.m`. When it finds one, it loads the file and executes it. Binary MEX-files take precedence over M-files when like-named files exist in the same directory. However, help text documentation still reads from the `.m` file.

You cannot use a binary MEX-file on a platform if it was compiled on a different platform. You must recompile the source code on the platform for which you want to use the MEX-file.

Binary MEX-File Placement

For MATLAB to be able to execute your C or Fortran functions, you must either put the compiled MEX-files containing those functions in a directory on the MATLAB path, or run MATLAB in the directory in which they reside. Functions in the current working directory are found before functions on the MATLAB path.

Type `path` to see what directories are currently included in your path. You can add new directories to the path either by using the `addpath` function, or by selecting **File > SetPath** to edit the path.

If you are using a Windows operating system and any of your binary MEX-files are on a network drive, be aware that file servers do not always report directory and file changes correctly. If you change any MEX-files that are on a network drive and you find that MATLAB is not using your latest changes, you can force MATLAB to look for the correct version of the file by changing directories away from and then back to the directory in which the files reside.

The Distinction Between mx and mex Prefixes

Routines in the MATLAB C and Fortran API that are prefixed with `mx` allow you to create, access, manipulate, and destroy `mxArrays`. Routines prefixed with `mex` perform operations back in the MATLAB environment.

mx Routines

The array access and creation library provides a set of array access and creation routines for manipulating MATLAB arrays. These subroutines, which are fully documented in the online API reference pages, always start with the prefix `mx`. For example, `mxGetPi` retrieves the pointer to the imaginary data inside the array.

Although most of the routines in the array access and creation library let you manipulate the MATLAB array, there are two exceptions—the IEEE® routines and memory management routines. For example, `mxGetNaN` returns a `double`, not an `mxArray`.

mex Routines

Routines that begin with the `mex` prefix perform operations back in the MATLAB environment. For example, the `mexEvalString` routine evaluates a string in the MATLAB workspace.

Note `mex` routines are only available in MEX-functions.

MATLAB Data

The MATLAB Array

The MATLAB language works with only a single object type: the MATLAB array. All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, structures, and objects, are stored as MATLAB arrays. In C, the MATLAB array is declared to be of type mxArray. The mxArray structure contains, among other things:

- Its type

- Its dimensions

- The data associated with this array

- If numeric, whether the variable is real or complex

- If sparse, its indices and nonzero maximum elements

- If a structure or object, the number of fields and field names

Data Storage

All MATLAB data is stored columnwise, which is how Fortran stores matrices. MATLAB uses this convention because it was originally written in Fortran. For example, given the matrix:

```
a=['house'; 'floor'; 'porch']
a =
   house
   floor
   porch
```

its dimensions are:

```
size(a)
ans =
     3     5
```

and its data is stored as

h	f	p	o	l	o	u	o	r	s	o	c	e	r	h

MATLAB Types

Complex Double-Precision Matrices

The most common data type in MATLAB is the complex double-precision, nonsparse matrix. These matrices are of type double and have dimensions m-by-n, where m is the number of rows and n is the number of columns. The data is stored as two vectors of double-precision numbers—one contains the real data and one contains the imaginary data. The pointers to this data are referred to as pr (pointer to real data) and pi (pointer to imaginary data), respectively. A real-only, double-precision matrix is one whose pi is NULL.

Numeric Matrices

MATLAB also supports other types of numeric matrices. These are single-precision floating-point and 8-, 16-, and 32-bit integers, both signed and unsigned. The data is stored in two vectors in the same manner as double-precision matrices.

Logical Matrices

The logical data type represents a logical true or false state using the numbers 1 and 0, respectively. Certain MATLAB functions and operators return logical 1 or logical 0 to indicate whether a certain condition was found to be true or not. For example, the statement (5 * 10) > 40 returns a logical 1 value.

MATLAB Strings

MATLAB strings are of type `char` and are stored the same way as unsigned 16-bit integers except there is no imaginary data component. Unlike C, MATLAB strings are not null terminated.

Cell Arrays

Cell arrays are a collection of MATLAB arrays where each `mxArray` is referred to as a cell. This allows MATLAB arrays of different types to be stored together. Cell arrays are stored in a similar manner to numeric matrices, except the data portion contains a single vector of pointers to `mxArrays`. Members of this vector are called cells. Each cell can be of any supported data type, even another cell array.

Structures

A `1`-by-`1` structure is stored in the same manner as a `1`-by-`n` cell array where `n` is the number of fields in the structure. Members of the data vector are called fields. Each field is associated with a name stored in the `mxArray`.

Objects

Objects are stored and accessed the same way as structures. In MATLAB, objects are named structures with registered methods. Outside MATLAB, an object is a structure that contains storage for an additional class name that identifies the name of the object.

Multidimensional Arrays

MATLAB arrays of any type can be multidimensional. A vector of integers is stored where each element is the size of the corresponding dimension. The storage of the data is the same as matrices.

Empty Arrays

MATLAB arrays of any type can be empty. An empty `mxArray` is one with at least one dimension equal to zero. For example, a double-precision `mxArray` of type `double`, where m and n equal `0` and `pr` is `NULL`, is an empty array.

Sparse Matrices

Sparse matrices have a different storage convention from that of full matrices in MATLAB. The parameters `pr` and `pi` are still arrays of double-precision numbers, but these arrays contain only nonzero data elements. There are three additional parameters: `nzmax`, `ir`, and `jc`.

- `nzmax` is an integer that contains the length of `ir`, `pr`, and, if it exists, `pi`. It is the maximum possible number of nonzero elements in the sparse matrix.

- `ir` points to an integer array of length `nzmax` containing the row indices of the corresponding elements in `pr` and `pi`.

- `jc` points to an integer array of length `n+1`, where `n` is the number of columns in the sparse matrix. The `jc` array contains column index information. If the `j`th column of the sparse matrix has any nonzero elements, `jc[j]` is the index in `ir` and `pr` (and `pi` if it exists) of the first nonzero element in the `j`th column, and `jc[j+1] - 1` is the index of the last nonzero element in that column. For the `j`th column of the sparse matrix, `jc[j]` is the total number of nonzero elements in all preceding columns. The last element of the `jc` array, `jc[n]`, is equal to `nnz`, the number of nonzero elements in the entire sparse matrix. If `nnz` is less than `nzmax`, more nonzero entries can be inserted into the array without allocating additional storage.

Using Data Types

You can write source MEX-files, MAT-file applications, and engine applications in C that accept any data type supported by MATLAB. In Fortran, only the creation of double-precision n-by-m arrays and strings are supported. You can treat binary C and Fortran MEX-files exactly like M-functions.

Caution MATLAB does not check the validity of MATLAB data structures created in C or Fortran using one of the `mx` functions (e.g., `mxCreateStructArray`). Using invalid syntax to create a MATLAB data structure can result in unexpected behavior in your C or Fortran program.

The explore Example

There is an example source MEX-file included with MATLAB, called `explore.c`, that identifies the data type of an input variable. The source file for this example is in the *matlabroot*/extern/examples/mex directory, where *matlabroot* represents the top-level directory where MATLAB is installed on your system.

> **Note** In platform-independent discussions that refer to directory paths, this book uses the UNIX convention. For example, a general reference to the mex directory is *matlabroot*/extern/examples/mex.

For example, typing:

```
cd([matlabroot '/extern/examples/mex']);
x = 2;
explore(x);
```

produces this result:

```
-------------------------------------------------
Name: prhs[0]
Dimensions: 1x1
Class Name: double
-------------------------------------------------
 (1,1) = 2
```

`explore` accepts any data type. Try using `explore` with these examples:

```
explore([1 2 3 4 5])
explore 1 2 3 4 5
explore({1 2 3 4 5})
explore(int8([1 2 3 4 5]))
explore {1 2 3 4 5}
explore(sparse(eye(5)))
explore(struct('name', 'Joe Jones', 'ext', 7332))
explore(1, 2, 3, 4, 5)
```

Building MEX-Files

In this section...
"What You Need to Build MEX-Files" on page 3-22
"Selecting a Compiler on Windows Platforms" on page 3-22
"Selecting a Compiler on UNIX Platforms" on page 3-28
"Overview of Building the timestwo MEX-File" on page 3-32

What You Need to Build MEX-Files

You need a compiler and the `mex` function to build MEX-files. MATLAB software supports many compilers and provides computer configuration files, called *options files*, designed specifically for these compilers. For an up-to-date list of supported compilers, see `Technical Note 1601`.

On 32-bit Microsoft Windows platforms, MATLAB provides a C compiler, Lcc. To view Help on using the Lcc compiler, type:

```
winopen(fullfile(matlabroot, '\sys\lcc\bin\wedit.hlp'))
```

If you have multiple compilers installed on your system, you can choose which compiler to use, as described in "Selecting a Compiler on Windows Platforms" on page 3-22 or "Selecting a Compiler on UNIX Platforms" on page 3-28.

To help you configure your system using a sample MEX-file, see "Overview of Building the timestwo MEX-File" on page 3-32.

If you have difficulty creating MEX-files, see "Creating a Source MEX-File" on page 3-4, or refer to "Troubleshooting MEX-Files" on page 3-47.

Selecting a Compiler on Windows Platforms

A *selected compiler configuration* specifies the compiler and build options MATLAB uses every time you invoke the `mex` build script. The compiler in this configuration is the *selected* compiler. It is the program that compiles source code into object code. A *configuration* is the set of programs and instructions that builds source code into shared libraries and standalone executable files.

To select a configuration, use the `mex -setup` command. You can set or change the configuration anytime, from either the MATLAB or the system command prompt. After you choose a configuration, it becomes the default and you no longer have to select one to compile MEX-files.

You can view information about the selected compiler configuration using the `mex.getCompilerConfigurations` function.

You can change the compiler configuration for a single call to the `mex` script using the `-f` switch, which specifies an options file. Subsequent calls to `mex` continue to use the selected compiler configuration.

For more information about these topics, see:

- "Viewing Supported Windows Compilers" on page 3-23
- "Selecting a Windows Compiler Configuration" on page 3-24
- "Getting Windows Configuration Information" on page 3-26
- "Specifying a Windows Options File" on page 3-27

Viewing Supported Windows Compilers

To see the list of supported compilers on the Windows platform, type:

```
mex -setup
```

MATLAB displays the following dialog. The text has been formatted to fit the page.

Note The list of compilers shown in your version of MATLAB may be different from the list shown in this example. For an up-to-date list of supported compilers, see `Technical Note 1601`.

```
Please choose your compiler for building external interface (MEX) files:

Would you like mex to locate installed compilers [y]/n? N

Select a compiler:
```

```
[1] Intel C++ 9.1 (with Microsoft Visual C++ 2005 linker)
[2] Intel Visual Fortran 10.1 (with Microsoft Visual C++ 2005 linker)
[3] Intel Visual Fortran 9.1 (with Microsoft Visual C++ 2005 linker)
[4] Lcc-win32 C 2.4.1
[5] Microsoft Visual C++ 6.0
[6] Microsoft Visual C++ .NET 2003
[7] Microsoft Visual C++ 2005
[8] Microsoft Visual C++ 2005 Express Edition
[9] Microsoft Visual C++ 2008
[10] Open WATCOM C++
[11] Open WATCOM C++ 1.3

[0] None

Compiler: 0

Done . . .
```

Selecting a Windows Compiler Configuration

MATLAB helps you choose a compiler configuration by generating a list of
either:

- All supported compilers. This is the same information found in Technical
  Note 1601. To see this list, follow the instructions in "Viewing Supported
  Windows Compilers" on page 3-23.

- Installed compilers found on your system. Only compilers supported by
  MATLAB are in this list.

To select a configuration from a list of supported compilers found on your
system, type:

```
mex -setup
```

MATLAB displays the following dialog. The text has been formatted to fit
the page.

> **Note** The list of compilers shown on your system may be different from the list shown in this example. The path names to your compilers may also be different. For an up-to-date list of supported compilers, see `Technical Note 1601`.

```
Please choose your compiler for building external interface (MEX)
files.

Would you like mex to locate installed compilers [y]/n? y

Select a compiler:
[1] Intel Visual Fortran 9.1 (with Microsoft Visual C++ 2005 linker) in
    C:\Program Files\Intel\Compiler\Fortran\9.1
[2] Lcc-win32 C 2.4.1 in C:\PROGRA~1\MATLAB\R2007b\sys\lcc
[3] Microsoft Visual C++ 2005 in
    C:\Program Files\Microsoft Visual Studio 8

[0] None

Compiler: 2

Please verify your choices:

Compiler: Lcc-win32 C 2.4.1
Location: C:\PROGRA~1\MATLAB\R2007b\sys\lcc

Are these correct?([y]/n): y

Trying to update options file:
C:\WINNT\Profiles\hhowell\Application Data\MathWorks\MATLAB\R2007b\
   mexopts.bat
From template:
C:\PROGRA~1\MATLAB\R2007b\bin\win32\mexopts\lccopts.bat

Done . . .
```

When to Change the Selected Compiler Configuration. On Windows systems, if you create both C and Fortran MEX-files, you must choose the appropriate compiler for the language you are using. If your selected compiler is the wrong language, it generates error messages. To see the language of your selected compiler, type:

```
cc = mex.getCompilerConfigurations;
cc.Language
```

You can change the compiler using either `mex -setup` or by "Specifying a Windows Options File" on page 3-27.

Getting Windows Configuration Information

On Windows systems, there is one compiler configuration. Use the `mex.getCompilerConfigurations` function to find the selected compiler configuration.

To get information about the selected compiler, type:

```
cc = mex.getCompilerConfigurations
```

MATLAB creates a `mex.CompilerConfiguration` object `cc` and displays its properties:

```
cc =

mex.CompilerConfiguration
package: mex

properties:
            Name: 'Microsoft Visual C++ 2005'
    Manufacturer: 'Microsoft'
        Language: 'C++'
         Version: '8.0'
        Location: 'C:\Program Files\Microsoft Visual Studio 8'
         Details: [1x1 mex.CompilerConfigurationDetails]

list of methods
```

To see the build options used by the selected compiler, type:

```
ccOptions = cc.Details
```

MATLAB creates a `mex.CompilerConfigurationDetails` object `ccOptions` and displays the options:

```
ccOptions =

mex.CompilerConfigurationDetails
package: mex

properties:
          CompilerExecutable: 'cl'
               CompilerFlags: [1x120 char]
           OptimizationFlags: '/O2 /Oy- /DNDEBUG'
                   DebugFlags: '/Zi /Fd"%OUTDIR%%MEX_NAME%%MEX_EXT%.pdb"'
            LinkerExecutable: 'link'
                 LinkerFlags: [1x257 char]
    LinkerOptimizationFlags: ''
            LinkerDebugFlags: '/DEBUG
                               /PDB:"%OUTDIR%%MEX_NAME%%MEX_EXT%.pdb"'

list of methods
```

Specifying a Windows Options File

MATLAB includes template options files you can use with particular compilers. The options files are located in the following directories.

Platform	Directory
Windows	*matlabroot*\bin\win32\mexopts
Windows x64	*matlabroot*\bin\win64\mexopts

On Windows systems, the options file has a `.bat` file extension.

For information on how to modify options files for particular systems, see "Custom Building MEX-Files" on page 3-34.

Use the -f option to specify an options file. To use this option, at the MATLAB prompt, type:

```
mex filename -f optionsfile
```

where *optionsfile* is the full path to the options file.

You might need to specify an options file if you want to use a different compiler (and not use the -setup option), or you want to compile MAT or engine standalone programs.

Selecting a Compiler on UNIX Platforms

A *selected compiler configuration* specifies the compiler and build options MATLAB uses every time you invoke the mex build script. The compiler in this configuration is the *selected* compiler. It is the program that compiles source code into object code. A *configuration* is the set of programs and instructions that builds source code into shared libraries and standalone executable files.

To select a configuration, use the mex -setup command. You can set or change the configuration anytime, from either the MATLAB or the system command prompt. After you choose a configuration, it becomes the default and you no longer have to select one to compile MEX-files.

You can view information about the selected compiler configuration using the mex.getCompilerConfigurations function.

You can change the compiler configuration for a single call to the mex script using the -f switch, which specifies an options file. Subsequent calls to mex continue to use the selected compiler configuration.

For more information about these topics, see:

- "Selecting a UNIX Compiler Configuration" on page 3-29
- "Getting UNIX Configuration Information" on page 3-30
- "Specifying a UNIX Options File" on page 3-31

Selecting a UNIX Compiler Configuration

You can set or change your compiler configuration anytime from either the
MATLAB command prompt or the UNIX shell, using the command:

```
mex -setup
```

MATLAB shows you the available compiler configurations, called options
files, in the following message:

```
Options files control which compiler to use, the compiler and link
command options, and the runtime libraries to link against.

Using the 'mex -setup' command selects an options file that is
placed in ~/.matlab/R2008a and used by default for 'mex'. An
options file in the current working directory or specified on the
command line overrides the default options file in
~/.matlab/R2008a.

To override the default options file, use the 'mex -f' command
(see 'mex -help' for more information).
```

MATLAB generates a list similar to the following:

```
The options files available for mex are:

  1: /mathworks/AH/devel/bat/Akernel/perfect/matlab/bin/f90opts.sh :
       Template Options file for building Fortran 90 MEX-files via the
 system ANSI compiler

  2: /mathworks/AH/devel/bat/Akernel/perfect/matlab/bin/gccopts.sh :
       Template Options file for building gcc MEX-files

  3: /mathworks/AH/devel/bat/Akernel/perfect/matlab/bin/mexopts.sh :
       Template Options file for building MEX-files via the system ANSI
 compiler

  0: Exit with no changes
```

To choose the compiler you want to use, respond to the MATLAB prompt:

```
Enter the number of the compiler (0-3):
```

Type the number corresponding to your selection. (If you do not want to change your configuration, type 0. MATLAB returns to the command prompt.) MATLAB displays information similar to the following:

```
matlabroot/bin/gccopts.sh is being copied to
/home/auser/.matlab/R2008a/mexopts.sh
```

Getting UNIX Configuration Information

On UNIX systems, there are three configurations, one for each compiler language (C, C++ and Fortran). Use the mex.getCompilerConfigurations function to view details about the compiler configurations.

To get information about the your compiler configuration, type:

```
cc = mex.getCompilerConfigurations
```

MATLAB creates a mex.CompilerConfiguration object cc and displays its properties:

```
cc =

1x3 mex.CompilerConfiguration
package: mex

properties:
    Name
    Manufacturer
    Language
    Version
    Location
    Details

list of methods
```

On the UNIX platform, cc is an array of three CompilerConfiguration objects – one for each language (C, C++, and Fortran). To see the compiler names, type:

```
disp('Compiler Name')
```

```
for i = 1:3; disp(cc(i).Name); end;
```

MATLAB displays:

```
Compiler Name
GNU C
GNU C++
g95
```

Note On UNIX systems, `mex.CompilerConfiguration.Location` is an empty string

Specifying a UNIX Options File

MATLAB includes template options files you can use with particular compilers. The options files are located in the *matlabroot*/bin directory.

The UNIX options file is named `*opts.sh`, where `*` is either `mex` or a specific compiler name.

For information on how to modify options files for particular systems, see "Custom Building MEX-Files" on page 3-34.

Use the `-f` option to specify an options file. To use this option, at the MATLAB prompt, type:

```
mex filename -f optionsfile
```

where *optionsfile* is the full path to the options file.

You might need to specify an options file in the following situations:

- You want to use a different compiler (and not use the `-setup` option), or you want to compile MAT or engine standalone programs.

- You do not want to use the system C compiler.

Overview of Building the timestwo MEX-File

MATLAB provides an example MEX-file, timestwo, for you to use to configure your system. This function takes a scalar input and doubles it.

The C source file is timestwo.c, and the Fortran source file is timestwo.F. These files are in the *matlabroot*\extern\examples\refbook directory, where *matlabroot* is the MATLAB root directory, the value returned by the matlabroot command.

To work with these files, copy them to a local directory. For example:

```
cd('c:\work')
copyfile([matlabroot '\extern\examples\refbook\timestwo.c'])
copyfile([matlabroot '\extern\examples\refbook\timestwo.F'])
```

To select your compiler, follow the instructions in either "Selecting a Compiler on UNIX Platforms" on page 3-28 or "Selecting a Compiler on Windows Platforms" on page 3-22.

Use the mex function to build the binary MEX-file. If you are using a C compiler, type:

```
mex timestwo.c
```

If you are using a Fortran compiler, type:

```
mex timestwo.F
```

This command creates the file timestwo.*mexext*, where *mexext* is the value returned by the mexext function. You call timestwo as if it were an M-function. For example, at the MATLAB command prompt, type:

```
timestwo(4)
```

MATLAB displays:

```
ans =
    8
```

Note In a future version of MATLAB, the default `mex` function will change to use the large-array-handling API. This means the `-largeArrayDims` option will be the default. For information about `mex` options, see "MEX Script Switches" on page 3-34. For information about the large-array-handling API, see "Handling Large mxArrays" on page 4-27.

Custom Building MEX-Files

In this section...
"Who Should Read This Chapter" on page 3-34
"MEX Script Switches" on page 3-34
"UNIX Default Options File" on page 3-38
"Windows Default Options File" on page 3-39
"Custom Building on UNIX Systems" on page 3-39
"Custom Building on Windows Systems" on page 3-42

Who Should Read This Chapter

In general, the defaults that come with MATLAB software should be sufficient for building most binary MEX-files. Following are reasons that you might need more detailed information:

- You want to use an Integrated Development Environment (IDE), rather than the provided script, to build MEX-files.

- You want to create a new options file, for example, to use a compiler that is not directly supported.

- You want to exercise more control over the build process than the script uses.

The script, in general, uses two stages (or three, for Microsoft Windows platforms) to build MEX-files. These are the compile stage and the link stage. In between these two stages, Windows compilers must perform some additional steps to prepare for linking (the prelink stage).

MEX Script Switches

The mex script has a set of switches (also called **options**) that you can use to modify the link and compile stages. The MEX Script Switches table lists the

available switches and their uses. Each switch is available on both UNIX[7] and Windows systems unless otherwise noted.

For customizing the build process, you should modify the options file, which contains the compiler-specific flags corresponding to the general compile, prelink, and link steps required on your system. The options file consists of a series of variable assignments; each variable represents a different logical piece of the build process.

MEX Script Switches

Switch	Function
@*rsp_file*	(Windows systems only) Include the contents of the text file *rsp_file* as command-line arguments to `mex`.
-*arch*	Build an output file for architecture *arch*. To determine the value for *arch*, type `computer('arch')` at the MATLAB Command Prompt on the target machine. Valid values for *arch* depend on the architecture of the build platform.
-ada *sfcn.ads*	Use this option to compile a Simulink® S-function written in Ada, where *sfcn.ads* is the Package Specification for the S-function. When this option is specified, only the -v (verbose) and -g (debug) options are relevant. All other options are ignored. For examples and information on supported compilers and other requirements, see `README` in the `simulink/ada/examples` directory.
-argcheck	(C functions only) Add argument checking. This adds code so arguments passed incorrectly to MATLAB API functions cause assertion failures.
-c	Compile only. Creates an object file, but not a binary MEX-file.

---

7. UNIX is a registered trademark of The Open Group in the United States and other countries.

MEX Script Switches (Continued)

Switch	Function
-compatibleArrayDims	Build a binary MEX-file using the MATLAB Version 7.2 array-handling API, which limits arrays to 2^31-1 elements. This option is the default, but in the future the -largeArrayDims option will be the default.
-cxx	(UNIX systems only) Use the C++ linker to link the MEX-file if the first source file is in C and there are one or more C++ source or object files. This option overrides the assumption that the first source file in the list determines which linker to use.
-D*name*	Define a symbol name to the C preprocessor. Equivalent to a #define *name* directive in the source.
-D*name*=*value*	Define a symbol name and value to the C preprocessor. Equivalent to a #define *name* *value* directive in the source.
-f *optionsfile*	Specify location and name of options file to use. Overrides the mex default-options-file search mechanism.
-fortran	(UNIX systems only) Specify that the gateway routine is in Fortran. This option overrides the assumption that the first source file in the list determines which linker to use.
-g	Create a binary MEX-file containing additional symbolic information for use in debugging. This option disables the mex default behavior of optimizing built object code (see the -O option).
-h[elp]	Print help for mex.
-I*pathname*	Add *pathname* to the list of directories to search for #include files.

MEX Script Switches (Continued)

Switch	Function
-inline	Inline matrix accessor functions (mx*). The generated MEX-function may not be compatible with future versions of MATLAB.
-l*name*	Link with object library. On Windows systems, *name* expands to *name*.lib or lib*name*.lib and on UNIX systems, to lib*name*.so or lib*name*.dylib.
-L*directory*	Add *directory* to the list of directories to search for libraries specified with the -l option. On UNIX systems, you must also set the run-time library path, as explained in "Setting Run-Time Library Path" on page 1-15.
-largeArrayDims	Build a binary MEX-file using the MATLAB large-array-handling API. This API can handle arrays with more than $2^{31}-1$ elements when compiled on 64-bit platforms. (See also the -compatibleArrayDims option.)
-n	No execute mode. Print any commands that mex would otherwise have executed, but do not actually execute any of them.
-O	Optimize the object code. Optimization is enabled by default and by including this option on the command line. If the -g option appears without the -O option, optimization is disabled.
-outdir *dirname*	Place all output files in directory *dirname*.
-output *resultname*	Create binary MEX-file named *resultname*. Automatically appends the appropriate MEX-file extension. Overrides the default MEX-file naming mechanism.
-setup	Specify the compiler options file to use when calling the mex function. When you use this option, all other command-line options are ignored.

MEX Script Switches (Continued)

Switch	Function
-U*name*	Remove any initial definition of the C preprocessor symbol *name*. (Inverse of the -D option.)
-v	Verbose mode. Print the values for important internal variables after the options file is processed and all command-line arguments are considered. Prints each compile step and final link step fully evaluated.
name=value	Override an options file variable for variable *name*. For examples, see Override Option Details in the Remarks section of the mex reference page.

UNIX Default Options File

The default MEX options file provided with MATLAB is located in *matlabroot*/bin. The mex script searches for an options file called mexopts.sh in the following order:

- The current directory

- The directory specified by *matlabroot*/bin

- The directory returned by the prefdir function

mex uses the first occurrence of the options file it finds. If no options file is found, mex displays an error message. You can directly specify the name of the options file using the -f switch.

The UNIX options file is written in the Bourne shell script language.

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file in fullfile(*matlabroot*, 'bin', 'mexopts.sh'), or you can invoke the mex script in verbose mode (-v). Verbose mode prints the exact compiler options, prelink commands (if appropriate), and linker options used in the build process for each compiler. "Custom Building on UNIX Systems" on page 3-39 gives an overview of the high-level build process.

Windows Default Options File

The default MEX options file is placed in your `user profile` directory after you configure your system by running `mex -setup`. The `mex` script searches for an options file called `mexopts.bat` in the following order:

- The current directory
- The user profile directory (returned by the `prefdir` function)

`mex` uses the first occurrence of the options file it finds. If no options file is found, `mex` searches your machine for a supported C compiler and automatically configures itself to use that compiler. Also, during the configuration process, it copies the compiler's default options file to the `user profile` directory. If multiple compilers are found, you are prompted to select one.

On Windows systems, the options file is written in the Perl script language.

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file, `mexopts.bat`, or you can invoke the `mex` script in verbose mode (`-v`). Verbose mode prints the exact compiler options, prelink commands, if appropriate, and linker options used in the build process for each compiler. "Custom Building on Windows Systems" on page 3-42 gives an overview of the high-level build process.

The User Profile Directory

The Windows `user profile` directory is a directory that contains user-specific information such as desktop appearance, recently used files, and **Start** menu items. The `mex` and `mbuild` utilities store their respective options files, `mexopts.bat` and `compopts.bat`, which are created during the `-setup` process, in a subdirectory of your `user profile` directory, named `Application Data\MathWorks\MATLAB`.

Custom Building on UNIX Systems

On UNIX systems, there are two stages in MEX-file building: compiling and linking.

Compile Stage

The compile stage must

- Add *matlabroot*/extern/include to the list of directories in which to find header files (-I*matlabroot*/extern/include).

- Define the preprocessor macro MATLAB_MEX_FILE (-DMATLAB_MEX_FILE).

- (C source MEX-files only) Compile the source file, which contains version information for the MEX-file, *matlabroot*/extern/src/mexversion.c.

Link Stage

The link stage must

- Instruct the linker to build a shared library.

- If you link with your own libraries, set the run-time library path, which is explained in "Setting Run-Time Library Path" on page 1-15.

- Link all objects from compiled source files (including mexversion.c).

- (Fortran source MEX-files only) Link in the precompiled versioning source file, *matlabroot*/extern/lib/$Arch/version4.o.

- Export the symbols mexFunction and mexVersion (these symbols represent functions called by MATLAB).

For Fortran MEX-files, the symbols are all lowercase and may have appended underscores. For specific information, invoke the mex script in verbose mode and examine the output.

Build Options

For customizing the build process, you should modify the options file. The options file contains the compiler-specific flags corresponding to the general steps outlined above. The options file consists of a series of variable assignments. Each variable represents a different logical piece of the build process. The options files provided with MATLAB are located in *matlabroot*/bin. The section "UNIX Default Options File" on page 3-38, describes how the mex script looks for an options file.

To aid in providing flexibility, there are two sets of options in the options file that you can turn on and off with switches to the mex script. These sets of options correspond to building in *debug mode* and building in *optimization mode*. They are represented by the variables DEBUGFLAGS and OPTIMFLAGS, respectively, one pair for each *driver* that is invoked (CDEBUGFLAGS for the C compiler, FDEBUGFLAGS for the Fortran compiler, and LDDEBUGFLAGS for the linker; similarly for the OPTIMFLAGS):

- If you build in optimization mode (the default), the mex script includes the OPTIMFLAGS options in the compile and link stages.

- If you build in debug mode, the mex script includes the DEBUGFLAGS options in the compile and link stages. It does not include the OPTIMFLAGS options.

- You can include both sets of options by specifying both the optimization and debugging flags to the mex script (-O and -g, respectively).

Aside from these special variables, the mex options file defines the executable invoked for each of the three modes (C compile, Fortran compile, link) and the flags for each stage. You also can provide explicit lists of libraries that must be linked in to all MEX-files containing source files of each language.

The variable summary follows.

Variable	C Compiler	Fortran Compiler	Linker
Executable	CC	FC	LD
Flags	CFLAGS	FFLAGS	LDFLAGS
Optimization	COPTIMFLAGS	FOPTIMFLAGS	LDOPTIMFLAGS
Debugging	CDEBUGFLAGS	FDEBUGFLAGS	LDDEBUGFLAGS
Additional libraries	CLIBS	FLIBS	(none)

For specifics on the default settings for these variables, you can

- Examine the options file in *matlabroot*/bin/mexopts.sh (or the options file you are using), or

• Invoke the `mex` script in verbose mode.

Custom Building on Windows Systems

There are three stages to MEX-file building for both C and Fortran on Windows systems: compiling, prelinking, and linking.

Compile Stage

For the compile stage, a `mex` options file must

• Set up paths to the compiler using the `COMPILER` (e.g., Watcom), `PATH`, `INCLUDE`, and `LIB` environment variables. If your compiler always has the environment variables set (e.g., in `AUTOEXEC.BAT`), you can comment them out in the options file.

• Define the name of the compiler, using the `COMPILER` environment variable, if needed.

• Define the compiler switches in the `COMPFLAGS` environment variable:

  - The switch to create a DLL is required for MEX-files.

  - For standalone programs, the switch to create an `exe` is required.

  - The `-c` switch (compile only; do not link) is recommended.

  - The switch to specify 8-byte alignment.

  - You can use any other switch specific to the environment.

• Define preprocessor `macro`, with `-D`, `MATLAB_MEX_FILE` is required.

• Set up optimizer switches and/or debug switches using `OPTIMFLAGS` and `DEBUGFLAGS`.

  - If you build in optimization mode (the default), the `mex` script includes the `OPTIMFLAGS` option in the compile stage.

  - If you build in debug mode, the `mex` script includes the `DEBUGFLAGS` options in the compile stage. It does not include the `OPTIMFLAGS` option.

  - You can include both sets of options by specifying both the optimization and debugging flags to the `mex` script (`OPTIMFLAGS` and `DEBUGFLAGS`, respectively).

Prelink Stage

The prelink stage dynamically creates import libraries to import the required function into the MEX, MAT, or engine file:

- All MEX-files link against `libmex.dll` (MEX library).

- MAT standalone programs link against `libmx.dll` (array access library) and `libmat.dll` (MAT-functions).

- Engine standalone programs link against `libmx.dll` (array access library) and `libeng.dll` for engine functions.

Link Stage

For the link stage, a `mex` options file must

- Define the name of the linker in the `LINKER` environment variable.

- Define the `LINKFLAGS` environment variable that must contain

  - The switch to create a DLL for MEX-files, or the switch to create an `exe` for standalone programs.

  - Export of the entry point to the MEX-file as `mexFunction` for C or `MEXFUNCTION@16` for Fortran.

  - The import library (or libraries) created in the `PRELINK_CMDS` stage.

  - You can use any other link switch specific to the compiler.

- Set up the linking optimization and debugging switches `LINKOPTIMFLAGS` and `LINKDEBUGFLAGS`. Use the same conditions described in the "Compile Stage" on page 3-42.

- Define the link-file identifier in the `LINK_FILE` environment variable, if necessary. For example, Watcom uses `file` to identify that the name following is a file and not a command.

- Define the link-library identifier in the `LINK_LIB` environment variable, if necessary. For example, Watcom uses `library` to identify the name following is a library and not a command.

- Optionally, set up an output identifier and name with the output switch in the `NAME_OUTPUT` environment variable. The environment variable `MEX_NAME` contains the name of the first program in the command line. This

must be set for `-output` to work. If this environment is not set, the compiler default is to use the name of the first program in the command line. Even if this is set, you can override it by specifying the `mex -output` switch.

Linking DLL Files to Binary MEX-Files

To link a DLL to a MEX-file, list the DLL's `.lib` file on the command line.

Versioning Build MEX-Files

The `mex` build script can build your MEX-file with a resource file that contains versioning and other essential information. The resource file is called `mexversion.rc` and resides in the `extern\include` directory. To support versioning, there are two new commands in the options files, `RC_COMPILER` and `RC_LINKER`, to provide the resource compiler and linker commands. It is assumed that

- If a compiler command is given, the compiled resource links to the MEX-file using the standard link command.

- If a linker command is given, the resource file links to the MEX-file after it is built using that command.

Compiling MEX-Files with the Microsoft Visual C++ IDE

Note This section provides information on how to compile source MEX-files in the Microsoft Visual C++ IDE. It is not totally inclusive. This section assumes that you know how to use the IDE. If you need more information on using the Microsoft Visual C++ IDE, refer to the corresponding Microsoft documentation.

To build MEX-files with the Microsoft Visual C++ integrated development environment:

1 Create a project and insert your MEX source files.

2 Add `mexversion.rc` from the MATLAB `include` directory, `matlab\extern\include`, to the project.

3 Create a `.def` file to export the MEX entry point. On the **Project** menu, click **Add New Item** and select **Module-Definition File (.def)**. For example:

```
LIBRARY MYFILE
EXPORTS mexFunction          <-- for a C MEX-file
    or
EXPORTS _MEXFUNCTION@16      <-- for a Fortran MEX-file
```

4 On the **Project** menu, click **Properties** for the project to open the property pages.

5 Under C/C++ General properties, add the MATLAB `include` directory, `matlab\extern\include`, as an additional include directory.

6 Under C/C++ Preprocessor properties, add `MATLAB_MEX_FILE` as a preprocessor definition.

7 Under Linker General properties, change the output file extension to `.mexw32` if you are building for a 32–bit platform or `.mexw64` if you are building for a 64–bit platform.

8 Locate the `.lib` files for the compiler you are using under *matlabroot*`\extern\lib\win32\microsoft` or *matlabroot*`\extern\lib\win64\microsoft`. Under Linker Input properties, add `libmx.lib`, `libmex.lib`, and `libmat.lib` as additional dependencies.

9 Under **Linker Input properties**, add the module definition (`.def`) file you created.

10 Under **Linker Debugging properties**, if you intend to debug the MEX-file using the IDE, specify that the build should generate debugging information. For more information about debugging, see "Debugging on the Microsoft Windows Platforms" on page 4-48.

If you are using a compiler other than the Microsoft Visual C++ compiler, the process for building MEX files is similar to that described above. In step 4, locate the `.lib` files for the compiler you are using in a subdirectory of *matlabroot*`\extern\lib\win32` or *matlabroot*`\extern\lib\win64`.

For example, if you are using an Open Watcom C++ compiler, look in
matlabroot\extern\lib\win32\watcom.

Troubleshooting MEX-Files

In this section...
"Configuration Issues" on page 3-47
"Understanding MEX-File Problems" on page 3-50
"Compiler and Platform-Specific Issues" on page 3-54
"Memory Management Issues" on page 3-55

Configuration Issues

This section focuses on common problems that might occur when creating binary MEX-files.

- "Search Path Problem on Microsoft Windows Systems" on page 3-47
- "MATLAB Path Names Containing Spaces on Windows Systems" on page 3-47
- "DLL Files Not on Path on MicrosoftWindows Systems" on page 3-48
- "Internal Error When Using mex -setup ()" on page 3-48
- "General Configuration Problem" on page 3-49

Search Path Problem on Microsoft Windows Systems

On Windows systems, if you move the MATLAB executable without reinstalling the MATLAB software, you may need to modify `mex.bat` to point to the new MATLAB location.

MATLAB Path Names Containing Spaces on Windows Systems

If you have problems building MEX-files on Windows systems and there is a space in any of the directory names within the MATLAB path, either reinstall MATLAB into a path name that contains no spaces or rename the directory that contains the space. For example, if you install MATLAB under the `Program Files` directory, you may have difficulty building MEX-files with certain C compilers.

DLL Files Not on Path on MicrosoftWindows Systems

MATLAB fails to load binary MEX-files if it cannot find all `.dll` files referenced by the MEX-file; the `.dll` files must be on the DOS path or in the same directory as the MEX-file. This is also true for third-party `.dll` files.

When this happens, MATLAB displays an error message of the following form:

```
??? Invalid MEX-file <mexfilename>:
The specified module could not be found.
```

On Windows systems, the third-party product Dependency Walker can be used to diagnose errors related to loading and executing modules. Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module and builds a hierarchical tree diagram of all dependent modules. For each module found, it lists all the functions that are exported by that module, and which of those functions are actually being called by other modules. You can download the Dependency Walker utility from the following Web site:

```
http://www.dependencywalker.com/
```

See the Technical Support solution 1-2RQL4L for information on using the Dependency Walker:

```
http://www.mathworks.com/support/solutions/data/1-2RQL4L.html
```

Internal Error When Using mex -setup ()

Some antivirus software packages may conflict with the `mex -setup` process or other `mex` commands. If you get an error message of the following form in response to a mex command:

```
mex.bat: internal error in sub get_compiler_info(): don't
recognize <string>
```

then you need to disable your antivirus software temporarily and reenter the command. After you have successfully run the `mex` script, you can reenable your antivirus software.

Alternatively, you can open a separate MS-DOS window and enter the `mex` command from that window.

General Configuration Problem

Make sure you followed the configuration steps for your platform described in this chapter. Also, refer to "Custom Building MEX-Files" on page 3-34 for additional information.

Understanding MEX-File Problems

This section contains information regarding common problems that occur when creating binary MEX-files. Use the following figure to help isolate these problems.
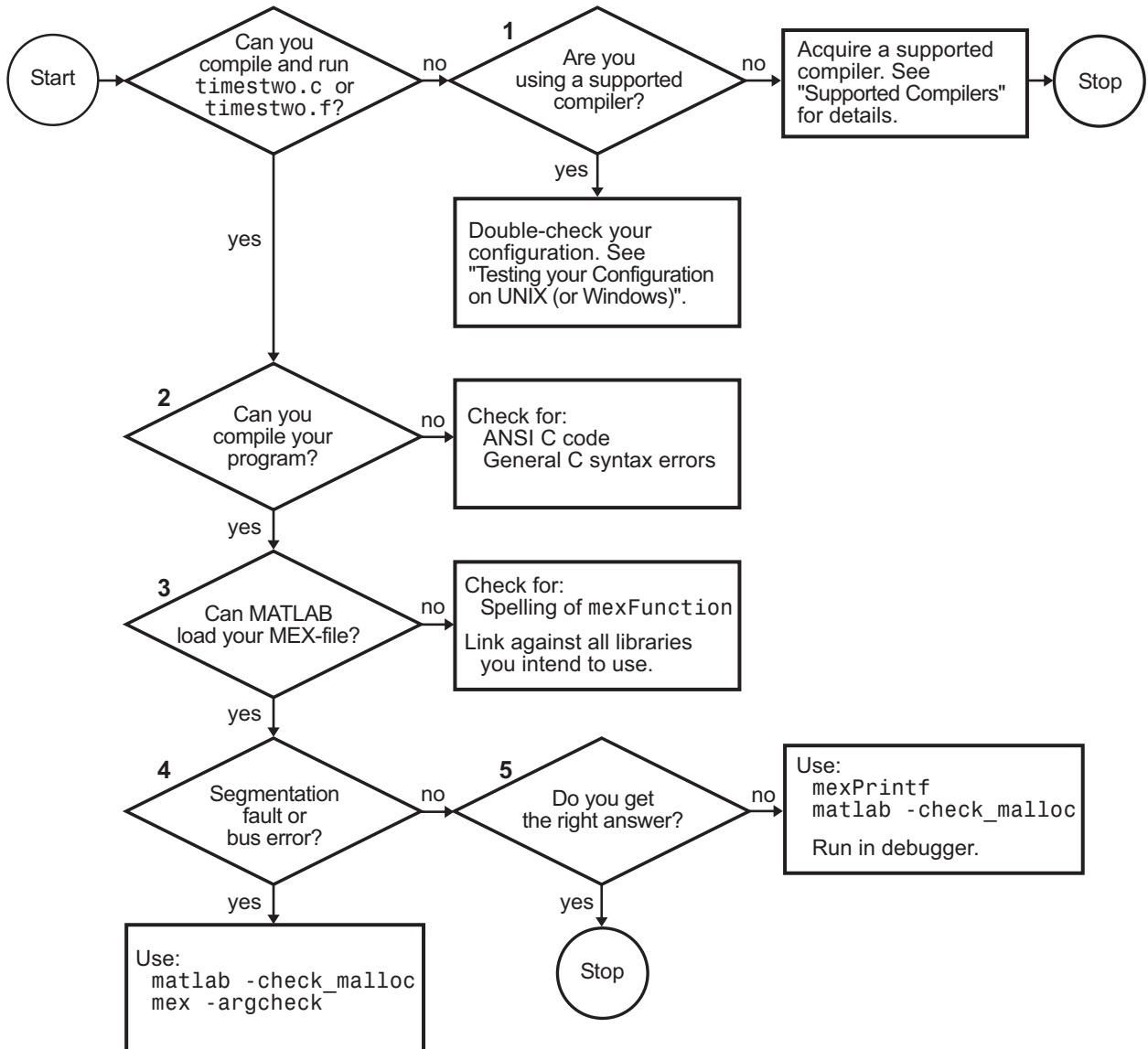
Troubleshooting MEX-File Creation Problems

Problems 1 through 5 refer to the corresponding numbered sections of the previous flowchart. For additional suggestions on resolving MEX-file build problems, access The MathWorks Technical Support Web site at:

```
http://www.mathworks.com/support
```

Problem 1 — Compiling a Source MEX-File Fails

The most common configuration problem in creating C source MEX-files on UNIX[8] systems involves using a non-ANSI C compiler, or failing to pass to the compiler a flag that tells it to compile ANSI C code.

A reliable way of knowing if you have this type of configuration problem is if the header files supplied by MATLAB generate a string of syntax errors when you try to compile your code. See "Building MEX-Files" on page 3-22 for information on selecting the appropriate options file or, if necessary, obtain an ANSI C compiler.

Problem 2 — Compiling Your Own Program Fails

Mixing ANSI and non-ANSI C code can generate a string of syntax errors. MATLAB provides header and source files that are ANSI C compliant. Therefore, your C code must also be ANSI compliant.

Other common problems that can occur in any C program are neglecting to include all necessary header files, or neglecting to link against all required libraries.

---

8. UNIX is a registered trademark of The Open Group in the United States and other countries.

Make sure you are using a MATLAB-supported compiler. See "What You Need to Build MEX-Files" on page 3-22 for this information. Additional information can be found in "Compiler and Platform-Specific Issues" on page 3-54.

Problem 3 — Binary MEX-File Load Errors

If you receive an error of the form:

```
Unable to load mex file:
??? Invalid MEX-file
```

MATLAB does not recognize your MEX-file.

MATLAB loads MEX-files by looking for the gateway routine, `mexFunction`. If you misspell the function name, MATLAB cannot load your MEX-file and generates an error message. On Windows systems, check that you are exporting `mexFunction` correctly.

On some platforms, if you fail to link against required libraries, you may get an error when MATLAB loads your MEX-file rather than when you compile your MEX-file. In such cases, a system error message referring to *unresolved symbols* or *unresolved references* appears. Be sure to link against the library that defines the function in question.

On Windows systems, MATLAB fails to load MEX-files if it cannot find all `.dll` files referenced by the MEX-file; the `.dll` files must be on the path or in the same directory as the MEX-file. This is also true for third-party `.dll` files. See "DLL Files Not on Path on MicrosoftWindows Systems" on page 3-48 for information to diagnose this problem.

Problem 4 — Segmentation Fault or Bus Error

If your binary MEX-file causes a segmentation violation or bus error, it means the MEX-file has attempted to access protected, read-only, or unallocated memory. Since this is such a general category of programming errors, such problems are sometimes difficult to track down.

Segmentation violations do not always occur at the same point as the logical errors that cause them. If a program writes data to an unintended section of memory, an error may not occur until the program reads and interprets the

corrupted data. Consequently, a segmentation violation or bus error can occur after the MEX-file finishes executing.

MATLAB provides three features to help you troubleshoot problems of this nature. Listed in order of simplicity, they are as follows:

• Recompile your source MEX-file with argument checking (C MEX-files only). You can add a layer of error checking to your MEX-file by recompiling with the `mex` script flag `-argcheck`. This warns you about invalid arguments to both MATLAB MEX-file (`mex`) and matrix access (`mx`) API functions.

  Although your MEX-file will not run as efficiently as it can, this switch detects errors such as passing `null` pointers to API functions.

• Run MATLAB with the `-check_malloc` option. The MATLAB startup flag, `-check_malloc`, indicates that MATLAB should maintain additional memory-checking information. When memory is freed, MATLAB checks to make sure that memory just before and just after this memory remains unwritten and that the memory has not been previously freed.

  If an error occurs, MATLAB reports the size of the allocated memory block. Using this information, you can track down where in your code this memory was allocated, and proceed accordingly.

  Although using this flag prevents MATLAB from running as efficiently as it can, it detects errors such as writing past the end of a dimensioned array, or freeing previously freed memory.

• Run MATLAB within a debugging environment. This process is already described in the chapters on creating C and Fortran source MEX-files, respectively.

Problem 5 - Program Generates Incorrect Results

If your program generates the wrong answer(s), there are several possible causes. First, there could be an error in the computational logic. Second, the program could be reading from an uninitialized section of memory. For example, reading the 11th element of a 10-element vector yields unpredictable results.

Another cause of generating a wrong answer could be overwriting valid data due to memory mishandling. For example, writing to the 15th element of a

10-element vector might overwrite data in the adjacent variable in memory. This case can be handled in a similar manner as segmentation violations, as described in Problem 4.

In all of these cases, you can use `mexPrintf` to examine data values at intermediate stages or run MATLAB within a debugger to exploit all the tools the debugger provides.

Compiler and Platform-Specific Issues

This section describes situations specific to particular compilers and platforms.

- "Using Binary MEX-Files from Other Sources" on page 3-54
- "Linux gcc Compiler Version Error" on page 3-54
- "Fortran Source MEX-Files Compiler Errors" on page 3-55
- "Binary MEX-Files Created in Watcom IDE" on page 3-55

Using Binary MEX-Files from Other Sources

If you obtain a binary MEX-file from another source, be sure the file was compiled for the same platform on which you want to run it. See "What Are MEX-Files?" on page 3-2 for platform-specific information.

When you try to run a binary MEX-file from a version of MATLAB that is different from the version that created the MEX-file, MATLAB displays an error message of the following form:

```
??? Invalid MEX-file <mexfilename>:
The specified module could not be found.
```

Linux gcc Compiler Version Error

For information concerning a gcc compiler version error on Linux systems, see the Technical Support solution 1-2H64MF at:

```
http://www.mathworks.com/support/solutions/data/1-2H64MF.html
```

Fortran Source MEX-Files Compiler Errors

When you try to compile a Fortran MEX-file using a free source form format, MATLAB displays an error message of the following form:

```
Illegal character in statement label field
```

`mex` supports the fixed source form. The difference between free and fixed source forms is explained in the Fortran Language Reference Manual Source Forms topic. The URL for this topic is:

```
http://h21007.www2.hp.com/portal/download/files/unprot/Fortran/
docs/lrm/lrm0015.htm#source_formatmenu?&Record=383697&STASH=7
```

The URL for the Fortran Language Reference Manual is:

```
http://h21007.www2.hp.com/portal/download/files/unprot/Fortran/
docs/lrm/dflrm.htm
```

Binary MEX-Files Created in Watcom IDE

If you use the Watcom IDE to create MEX-files and get unresolved references to API functions when linking against our libraries, check the argument-passing convention. The Watcom IDE uses a default switch that passes parameters in registers. MATLAB requires that you pass parameters on the stack.

Memory Management Issues

When a binary MEX-file returns control to MATLAB, it returns the results of its computations in the output arguments—the `mxArrays` contained in the left-hand side arguments `plhs[]`. MATLAB destroys any `mxArray` created by the MEX-file that is not in this argument list. In addition, MATLAB frees any memory that was allocated in the MEX-file using the `mxCalloc`, `mxMalloc`, or `mxRealloc` functions.

In general, we recommend that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism. This approach is consistent with other MATLAB API applications (i.e., MAT-file applications, engine applications,

and MATLAB® Compiler™ generated applications, which do not have any automatic cleanup mechanism.)

However, you should not destroy an `mxArray` in a source MEX-file when it is:

- passed to the MEX-file in the right-hand side list `prhs[]`
- returned in the left-hand side list `plhs[]`
- returned by `mexGetVariablePtr`
- used to create a structure

This section describes situations specific to memory management. We recommend you review code in your source MEX-files to avoid using these functions in the following situations. For additional information, see "Memory Management" on page 4-30 in Creating C Language MEX-Files. For guidance on memory issues, see "Strategies for Efficient Use of Memory". Additional tips are found in Technical Note 1107: "Avoiding Out of Memory Errors" at the following URL:

`http://www.mathworks.com/support/tech-notes/1100/1107.html`.

Potential memory management problems include:

- "Improperly Destroying an mxArray" on page 3-56
- "Incorrectly Constructing a Cell or Structure mxArray" on page 3-57
- "Creating a Temporary mxArray with Improper Data" on page 3-58
- "Creating Potential Memory Leaks" on page 3-58
- "Improperly Destroying a Structure" on page 3-59
- "Destroying Memory in a C++ Class Destructor" on page 3-60

Improperly Destroying an mxArray

Do not use `mxFree` to destroy an `mxArray`.

Example. In the following example, `mxFree` does not destroy the array object. This operation frees the structure header associated with the array, but MATLAB stills operates as if the array object needs to be destroyed. Thus MATLAB tries to destroy the array object, and in the process, attempts to free its structure header again:

```
mxArray *temp = mxCreateDoubleMatrix(1,1,mxREAL);
      ...
mxFree(temp);  /* INCORRECT */
```

Solution. Call `mxDestroyArray` instead:

```
mxDestroyArray(temp);  /* CORRECT */
```

Incorrectly Constructing a Cell or Structure mxArray

Do not call `mxSetCell` or `mxSetField` variants with `prhs[]` as the member array.

Example. In the following example, when the MEX-file returns, MATLAB destroys the entire cell array. Since this includes the members of the cell, this implicitly destroys the MEX-file's input arguments. This can cause several strange results, generally having to do with the corruption of the caller's workspace, if the right-hand side argument used is a temporary array (i.e., a literal or the result of an expression):

```
myfunction('hello')
/* myfunction is the name of your MEX-file and your code
/* contains the following:    */

    mxArray *temp = mxCreateCellMatrix(1,1);
      ...
    mxSetCell(temp, O, prhs[O]);  /* INCORRECT */
```

Solution. Make a copy of the right-hand side argument with `mxDuplicateArray` and use that copy as the argument to `mxSetCell` (or `mxSetField` variants). For example:

```
mxSetCell(temp, O, mxDuplicateArray(prhs[O]));  /* CORRECT */
```

Creating a Temporary mxArray with Improper Data

Do not call mxDestroyArray on an mxArray whose data was not allocated by an API routine.

Example. If you call mxSetPr, mxSetPi, mxSetData, or mxSetImagData, specifying memory that was not allocated by mxCalloc, mxMalloc, or mxRealloc as the intended data block (second argument), then when the MEX-file returns, MATLAB attempts to free the pointers to real data and imaginary data (if any). Thus MATLAB attempts to free memory, in this example, from the program stack:

```
mxArray *temp = mxCreateDoubleMatrix(0,0,mxREAL);
double data[5] = {1,2,3,4,5};
     ...
mxSetM(temp,1); mxSetN(temp,5); mxSetPr(temp, data);
/* INCORRECT */
```

Solution. Rather than use mxSetPr to set the data pointer, instead, create the mxArray with the right size and use memcpy to copy the stack data into the buffer returned by mxGetPr:

```
mxArray *temp = mxCreateDoubleMatrix(1,5,mxREAL);
double data[5] = {1,2,3,4,5};
     ...
memcpy(mxGetPr(temp), data, 5*sizeof(double)); /* CORRECT */
```

Creating Potential Memory Leaks

Prior to Version 5.2, if you created an mxArray using one of the API creation routines and then you overwrote the pointer to the data using mxSetPr, MATLAB still freed the original memory. This is no longer the case.

For example:

```
pr = mxCalloc(5*5, sizeof(double));
... <load data into pr>
plhs[0] = mxCreateDoubleMatrix(5,5,mxREAL);
mxSetPr(plhs[0], pr);  /* INCORRECT */
```

will now leak 5*5*8 bytes of memory, where 8 bytes is the size of a double.

You can avoid that memory leak by changing the code to:

```
plhs[0] = mxCreateDoubleMatrix(5,5,mxREAL);
pr = mxGetPr(plhs[0]);
... <load data into pr>
```

or alternatively:

```
pr = mxCalloc(5*5, sizeof(double));
... <load data into pr>
plhs[0] = mxCreateDoubleMatrix(5,5,mxREAL);
mxFree(mxGetPr(plhs[0]));
mxSetPr(plhs[0], pr);
```

Note that the first solution is more efficient.

Similar memory leaks can also occur when using mxSetPi, mxSetData, mxSetImagData, mxSetIr, or mxSetJc. You can avoid memory leaks by changing the code as described in this section.

Improperly Destroying a Structure

If you create a structure, you must call mxDestroyArray only on the structure. A field in the structure points to the data in the array used by mxSetField or mxSetFieldByNumber. When mxDestroyArray destroys the structure, it attempts to traverse down through itself and free all other data, including the memory in the data arrays. If you call mxDestroyArray on each data array, the same memory is freed twice and this can corrupt memory.

Example. The following example creates three arrays: one structure array aStruct and two data arrays, myDataOne and myDataTwo. Field name one contains a pointer to the data in myDataOne, and field name two contains a pointer to the data in myDataTwo.

```
mxArray *myDataOne;
mxArray *myDataTwo;
mxArray *aStruct;
const char *fields[] = { "one", "two" };

myDataOne = mxCreateDoubleScalar(1.0);
myDataTwo = mxCreateDoubleScalar(2.0);
```

```
aStruct = mxCreateStructMatrix(1,1,2,fields);
mxSetField( aStruct, 0, "one", myDataOne );
mxSetField( aStruct, 1, "two", myDataTwo );
mxDestroyArray(myDataOne);
mxDestroyArray(myDataTwo);
mxDestroyArray(aStruct);
```

Solution. The command `mxDestroyArray(aStruct)` destroys the data in all three arrays:

```
      ...
aStruct = mxCreateStructMatrix(1,1,2,fields);
mxSetField( aStruct, 0, "one", myDataOne );
mxSetField( aStruct, 1, "two", myDataTwo );
mxDestroyArray(aStruct);
```

Destroying Memory in a C++ Class Destructor

Do not use the `mxFree` or `mxDestroyArray` functions in a C++ destructor of a class used in a MEX-function. MATLAB performs cleanup of MEX-file variables if an error is thrown from the MEX-function, as described in "Automatic Cleanup of Temporary Arrays" on page 4-30.

If an error occurs that causes the object to go out of scope, the C++ destructor is called. Freeing memory directly in the destructor means both MATLAB and the destructor free the same memory, and this corrupts memory.

Additional Information for Using MEX-Files

In this section...
"Files and Directories — UNIX Operating Systems" on page 3-61
"Files and Directories — Microsoft Windows Operating Systems" on page 3-63
"Examples" on page 3-65
"Technical Support" on page 3-66

Files and Directories — UNIX Operating Systems

This section describes the directory organization and purpose of the files associated with the MATLAB C and Fortran API on UNIX[9] systems.

---
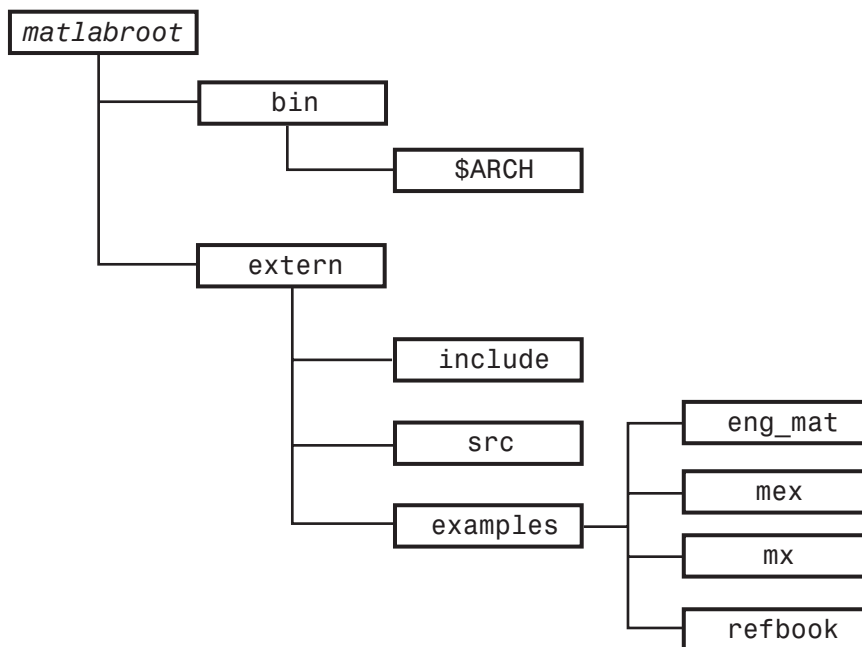
9. UNIX is a registered trademark of The Open Group in the United States and other countries.

matlabroot/bin

The *matlabroot*/bin directory contains two files that are relevant for the MATLAB API:

mex

    UNIX shell script that creates binary MEX-files from C or Fortran MEX-file source code.

matlab

    UNIX shell script that initializes your environment and then invokes the MATLAB interpreter.

This directory also contains the preconfigured options files that the mex script uses with particular compilers. For more information, see "Specifying a UNIX Options File" on page 3-31.

matlabroot/bin/$ARCH

The *matlabroot*/bin/*$ARCH* directory contains libraries, where *$ARCH* specifies a particular UNIX platform. On some UNIX platforms, this directory contains two versions of this library. Library file names ending with .so or .dylib are shared libraries.

matlabroot/extern/include

The *matlabroot*/extern/include directory contains the header files for developing C and C++ applications that interface with MATLAB.

The relevant header files for the MATLAB API are:

engine.h

    Header file for MATLAB engine programs. Contains function prototypes for engine routines.

mat.h

    Header file for programs accessing MAT-files. Contains function prototypes for mat routines.

matrix.h

    Header file containing a definition of the mxArray structure and function prototypes for matrix access routines.

`mex.h`

> Header file for building MEX-files. Contains function prototypes for `mex` routines.

matlabroot/extern/src

The *matlabroot*/extern/src directory contains those C source files that are necessary to support certain MEX-file features such as argument checking and versioning.

Files and Directories — Microsoft Windows Operating Systems

This section describes the directory organization and purpose of the files associated with the MATLAB C and Fortran API on Microsoft Windows systems.

The following figure illustrates the directories in which the MATLAB API files are located. In the illustration, *matlabroot* symbolizes the top-level directory where MATLAB is installed on your system.

```
matlabroot
    ├── bin
    │     └── win32
    │            └── mexopts
    └── extern
           ├── include
           ├── src
           └── examples
                    ├── eng_mat
                    ├── mex
                    ├── mx
                    └── refbook
```
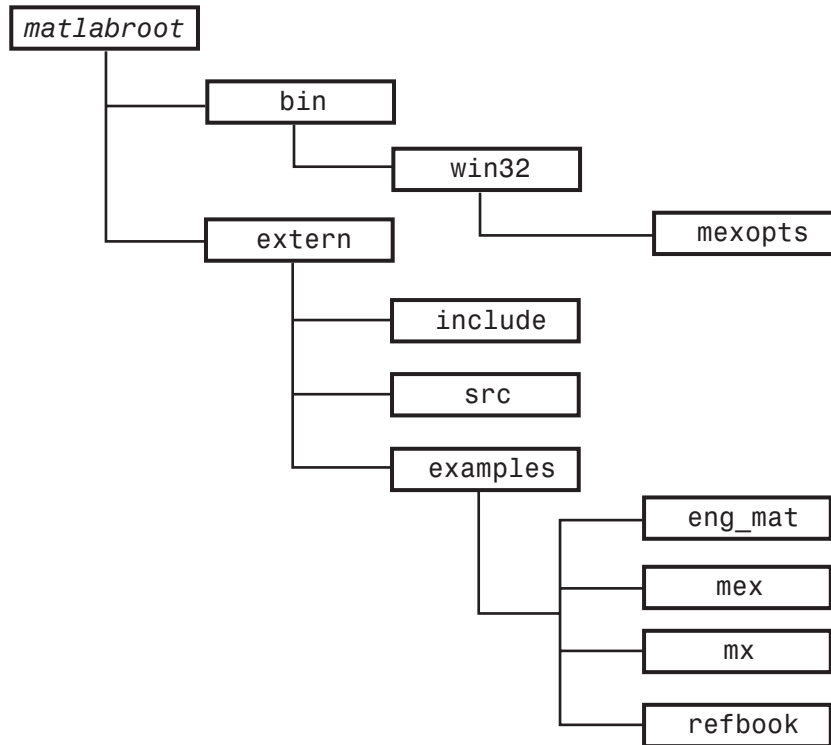
matlabroot\bin

The *matlabroot*\bin directory contains the mex.bat batch file that builds C and Fortran files into binary MEX-files. Also, this directory contains mex.pl, which is a Perl script used by mex.bat.

matlabroot\bin\win32\mexopts or matlabroot\bin\win64\mexopts

The *matlabroot*\bin\win32\mexopts or *matlabroot*\bin\win64\mexopts directory contains the preconfigured options files that the mex script uses

with particular compilers. For more information, see "Specifying a Windows Options File" on page 3-27.

matlabroot\extern\include

The *matlabroot*\extern\include directory contains the header files for developing C and C++ applications that interface with MATLAB.

The relevant header files for the MATLAB API (MEX-files, engine, and MAT-files) are

engine.h

> Header file for MATLAB engine programs. Contains function prototypes for engine routines.

mat.h

> Header file for programs accessing MAT-files. Contains function prototypes for mat routines.

matrix.h

> Header file containing a definition of the mxArray structure and function prototypes for matrix access routines.

mex.h

> Header file for building MEX-files. Contains function prototypes for mex routines.

*.def

> Files used by Microsoft Visual C++ and Microsoft Fortran compilers.

mexversion.rc

> Resource file for inserting versioning information into MEX-files.

matlabroot\extern\src

The *matlabroot*\extern\src directory contains files that are used for debugging MEX-files.

Examples

This book uses many examples to show how to write C and Fortran source MEX-files.

Examples from the Text

The `refbook` subdirectory in the `extern/examples` directory contains the MEX-file examples (C and Fortran) that are used in this topic.

MEX Reference Examples

The `mex` subdirectory of `/extern/examples` directory contains MEX-file examples. It includes the examples described in the online MATLAB C and Fortran API Reference for "MEX-Files" (the functions beginning with the `mex` prefix).

MX Examples

The `mx` subdirectory of `extern/examples` contains examples for using the array access functions. Although you can use these functions in standalone programs, most of these are MEX-file examples. The exception is `mxSetAllocFcns.c`, since this function is available only to standalone programs.

Engine and MAT Examples

The `eng_mat` subdirectory in the `extern/examples` directory contains the MEX-file examples (C and Fortran) for using the MATLAB engine facility, as well as examples for reading and writing MATLAB data files (MAT-files). These examples are all standalone programs.

Technical Support

The MathWorks provides additional Technical Support through its Web site. A few of the services provided are as follows:

• Solution Search Engine

This knowledge base on our Web site includes thousands of solutions and links to Technical Notes and is updated several times each week.

    http://www.mathworks.com/support/

• Technical Notes

Technical notes are written by our Technical Support staff to address commonly asked questions.

```
http://www.mathworks.com/support/tech-notes/list_all.shtml
```

Creating C Language MEX-Files

This chapter describes how to write source MEX-files in the C programming language. It discusses the source file itself, how these C language files interact with MATLAB software, how to pass and manipulate arguments of different data types, how to debug your binary MEX-file programs, and several other, more advanced topics.

- "C Source MEX-Files" on page 4-2
- "Examples of C Source MEX-Files" on page 4-11
- "Advanced Topics" on page 4-26
- "Debugging C Language MEX-Files" on page 4-48

C Source MEX-Files

In this section...
"The Components of a C MEX-File" on page 4-2
"Gateway Routine" on page 4-2
"Computational Routine" on page 4-5
"Preprocessor Macros" on page 4-5
"Data Flow in MEX-Files" on page 4-5
"Creating C++ MEX-Files" on page 4-9

The Components of a C MEX-File

You create binary MEX-files using the `mex` build script. `mex` compiles and links source files into a shared library called a binary MEX-file, which you can run at the MATLAB command line. Once compiled, you treat binary MEX-files exactly like MATLAB M-files and built-in functions.

This section explains the components of a source MEX-file, statements you use in a program source file. Unless otherwise specified, the term "MEX-file" refers to a source file.

The MEX-file consists of:

- A "Gateway Routine" on page 4-2 that interfaces C and MATLAB data.

- A "Computational Routine" on page 4-5 written in C that performs the computations you want implemented in the binary MEX-file.

- "Preprocessor Macros" on page 4-5 for building platform-independent code.

Gateway Routine

The *gateway routine* is the entry point to the MEX-file shared library. It is through this routine that MATLAB accesses the rest of the routines in your MEX-files. Use the following guideline to create a gateway routine:

- "Naming the Gateway Routine" on page 4-3

A C MEX-file gateway routine looks like this:

```
void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    /* more C code ... */
}
```

Naming the Gateway Routine

The name of the gateway routine must be `mexFunction`.

Required Parameters

A gateway routine must contain the parameters `prhs`, `nrhs`, `plhs`, and `nlhs` which are described in the following table.

Parameter	Description
prhs	An array of right-hand input arguments.
plhs	An array of left-hand output arguments.
nrhs	The number of right-hand arguments, or the size of the `prhs` array.
nlhs	The number of left-hand arguments, or the size of the `plhs` array.

Both `prhs` and `plhs` are declared as type `mxArray *`, which means they point to MATLAB arrays. They are vectors that contain pointers to the arguments of the MEX-file.

You can think of the name prhs as representing the "parameters, right-hand side," that is, the input parameters. Likewise, plhs represents the "parameters, left-hand side," or output parameters.

Creating and Using Source Files

It is good practice to write the gateway routine to call a "Computational Routine" on page 4-5; however, this is not required. The computational code can be part of the gateway routine. If you use both gateway and computational routines, they can be combined in one source file or in separate files. If you use separate files, the gateway routine must be the first source file listed in the mex command.

The name of the file containing your gateway routine is important, as explained in "Naming the MEX-File" on page 4-4.

Using MATLAB Libraries

The MATLAB C and Fortran API Reference describes functions you can use in your gateway and computational routines that interact with MATLAB programs and the data in the MATLAB workspace. The mx prefixed functions provide access methods for manipulating MATLAB arrays. The mex prefixed functions perform operations in the MATLAB environment.

Required Header Files

To use the functions in the C and Fortran Reference library you must include the mex header, which declares the entry point and interface routines. Put this statement in your source file:

```
#include "mex.h"
```

Naming the MEX-File

The binary MEX-file name, and hence the name of the function you use in MATLAB, is the name of the source file containing your gateway routine.

The file extension of the binary MEX-file is platform-dependent. You find the file extension using the mexext function, which returns the value for the current machine.

Computational Routine

The *computational routine* contains the code for performing the computations you want implemented in the binary MEX-file. Computations can be numerical computations as well as inputting and outputting data. The gateway calls the computational routine as a subroutine.

The programming requirements described in "Creating and Using Source Files" on page 4-4, "Using MATLAB Libraries" on page 4-4, and "Required Header Files" on page 4-4 may also apply to your computational routine.

Preprocessor Macros

The MATLAB *preprocessor macros* mwSize and mwIndex are used in the mx and mex functions for cross-platform flexibility. mwSize represents size values, such as array dimensions and number of elements. mwIndex represents index values, such as indices into arrays.

Data Flow in MEX-Files

The following examples illustrate data flow in MEX-files:

- "Showing Data Input and Output" on page 4-5
- "Gateway Routine Data Flow Diagram" on page 4-6
- "MATLAB Example yprime.c" on page 4-7

Showing Data Input and Output

Suppose your MEX-file myFunction has 2 input arguments and 1 output argument. The MATLAB syntax is [X] = myFunction(Y, Z). To call myFunction from MATLAB, type:

```
X = myFunction(Y, Z);
```

The MATLAB interpreter calls mexFunction, the gateway routine to myFunction, with the following arguments:

```
nlhs = 1

nrhs = 2
```

Your input is `prhs`, a 2-element C array (`nrhs = 2`). The first element is a pointer to an `mxArray` named `Y` and the second element is a pointer to an `mxArray` named `Z`.

Your output is `plhs`, a 1-element C array (`nlhs = 1`) where the single element is a `null` pointer. The parameter `plhs` points at nothing because the output `X` is not created until the subroutine executes.

The gateway routine creates the output array and sets a pointer to it in `plhs[0]`. If `plhs[0]` is left unassigned and you assign an output value to the function when you call it, MATLAB generates an error stating that no output was assigned.

Note It is possible to return an output value even if `nlhs = 0`. This corresponds to returning the result in the `ans` variable.

Gateway Routine Data Flow Diagram

The following MEX Cycle diagram shows how inputs enter a MEX-file, what functions the gateway routine performs, and how outputs return to MATLAB.

In this example, the syntax of the MEX-file `func` is `[C, D] = func(A,B)`. In the figure, a call to `func` tells MATLAB to pass variables `A` and `B` to your MEX-file. `C` and `D` are left unassigned.

The gateway routine `func.c` uses the `mxCreate*` functions to create the MATLAB arrays for your output arguments. It sets `plhs[0]` and `plhs[1]`

to the pointers to the newly created MATLAB arrays. It uses the mxGet*
functions to extract your data from your input arguments prhs[0] and
prhs[0]. Finally, it calls your computational routine, passing the input and
output data pointers as function parameters.

On return to MATLAB, plhs[0] is assigned to C and plhs[1] is assigned to D.

MATLAB

A call to
MEX-file func:
 [C,D]=func(A,B)

tells MATLAB to
pass variables A and
B to your MEX-file.
C and D are left
unassigned.

Inputs

```
const mxArray *B
B = prhs[1]
```

```
const mxArray *A
A = prhs[0]
```

func.c

```
void mexFunction(
   int nlhs, mxArray *plhs[],
   int nrhs, const mxArray *prhs[])
```

In the gateway routine:

- Use the mxCreate functions to create
  the MATLAB arrays for your output
  arguments. Set plhs[0],[1],...
  to the pointers to the newly created
  MATLAB arrays.

- Use the mxGet functions to extract
  your data from prhs[0],[1],....

- Call your C subroutine passing the
  input and output data pointers as
  function parameters.

MATLAB

On return from
MEX-file func:
 [C,D]=func(A,B)

plhs[0] is assigned
to C and plhs[1] is
assigned to D.

```
mxarray *D
D = plhs[1]
```

```
mxarray *C
C = plhs[0]
```

Outputs
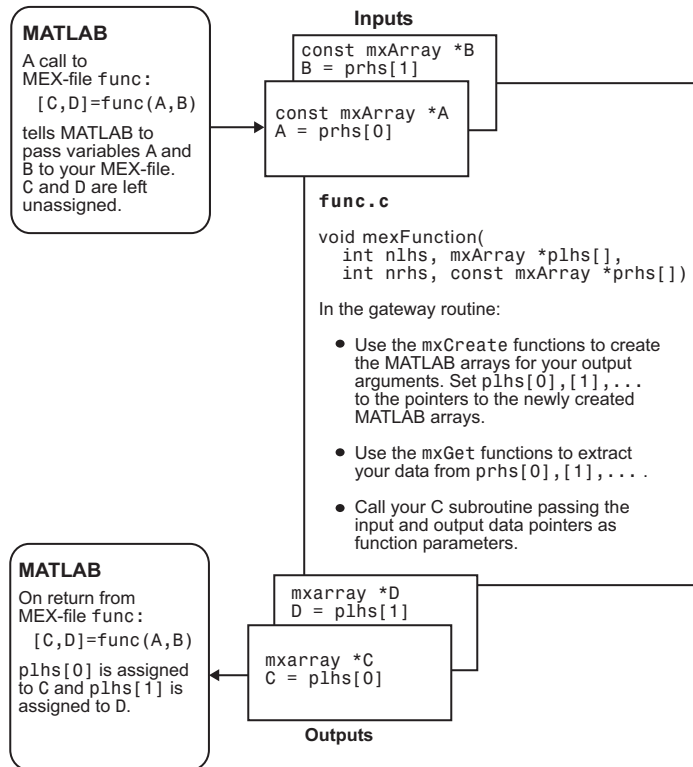
C MEX Cycle

MATLAB Example yprime.c

Let's look at an example, yprime.c, found in your
matlabroot/extern/examples/mex/ directory. ("Building MEX-Files" on
page 3-22 explains how to create the binary MEX-file.) It's calling syntax is
[YP] = YPRIME(T,Y), where T is an integer and Y is a vector with 4 elements.
For T=1 and Y=1:4, when you type:

```
    yprime(T,Y)
```

MATLAB displays:

```
ans =
    2.0000   8.9685   4.0000    -1.0947
```

The gateway routine should validate the input arguments. This step includes checking the number, type, and size of the input arrays as well as examining the number of output arrays. If the inputs are not valid, call mexErrMsgTxt. For example:

mexErrMsgTxt

```
/* Check for proper number of arguments */
if (nrhs != 2) {
 mexErrMsgTxt("Two input arguments required.");
} else if (nlhs > 1) {
 mexErrMsgTxt("Too many output arguments.");
}

/* Check the dimensions of Y.  Y can be 4 X 1 or 1 X 4. */
m = mxGetM(Y_IN);
n = mxGetN(Y_IN);
if (!mxIsDouble(Y_IN) || mxIsComplex(Y_IN) ||
    (MAX(m,n) != 4) || (MIN(m,n) != 1)) {
 mexErrMsgTxt("YPRIME requires that Y be a 4 x 1 vector.");
}
```

To create MATLAB arrays, call any of the mxCreate* functions, like mxCreateDoubleMatrix, mxCreateSparse, or mxCreateString. If it needs them, the gateway routine can call mxCalloc to allocate temporary work arrays for the computational routine. In this example:

```
/* Create a matrix for the return argument */
plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
```

In the gateway routine, you access the data in mxArray and manipulate it in your computational subroutine. For example, the expression mxGetPr(prhs[0]) returns a pointer of type double * to the real data in the

mxArray pointed to by `prhs[0]`.You can then use this pointer like any other pointer of type `double *` in C. For example,

```
/* Assign pointers to the various parameters */
yp = mxGetPr(plhs[0]);
```

In this example, a computational routine, `yprime`, performs the calculations:

```
/* Do the actual computations in a subroutine */
yprime(yp,t,y);
```

After calling your computational routine from the gateway, you can set a pointer of type `mxArray` to the data it returns. MATLAB recognizes the output from your computational routine as the output from the binary MEX-file.

When a binary MEX-file completes its task, it returns control to MATLAB. Any MATLAB arrays that are created by the MEX-file but are not returned to MATLAB through the left-hand side arguments are automatically destroyed.

In general, we recommend that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism.

Creating C++ MEX-Files

All C++ language standards are supported in MEX-files.

This section discusses specific C++ language issues to consider when creating and using MEX-files.

Creating Your C++ Source File

The C++ source code for the examples provided by MATLAB use the `.cpp` file extension. The extension `.cpp` is unambiguous and generally recognized by C++ compilers. Other possible extensions include `.C`, `.cc`, and `.cxx`.

For information on using C++ features, see Technical Note 1605, MEX-files Guide, at `http://www.mathworks.com/support/tech-notes/1600/1605.html`. Look for the sections under the "C++ Mex-files" heading.

Compiling and Linking

You can run a C++ MEX-file only on systems with the same version of MATLAB that the file was compiled on.

Use `mex   setup` to select a C++ compiler, then type:

```
mex filename.cpp
```

You can use command-line options, as shown in the "MEX Script Switches" on page 3-34 table.

Your link command must have all of the necessary DLL files that the MEX-function is dependent upon. To help you check for dependent files, see the Troubleshooting topic "DLL Files Not on Path on MicrosoftWindows Systems" on page 3-48.

Examples

The examples "Using C++ Features in MEX-Files" on page 4-22 and "File Handling with C++" on page 4-23 illustrate the use of C++ by walking through source code examples available in your MATLAB directory.

Memory Considerations For Class Destructors

Do not use the `mxFree` or `mxDestroyArray` functions in a C++ destructor of a class used in a MEX-function. MATLAB performs cleanup of MEX-file variables if an error is thrown from the MEX-function, as described in "Automatic Cleanup of Temporary Arrays" on page 4-30.

If an error occurs that causes the object to go out of scope, the C++ destructor is called. Freeing memory directly in the destructor means both MATLAB and the destructor free the same memory, and this corrupts memory.

Examples of C Source MEX-Files

Introduction

The MATLAB API provides a full set of routines that handle the various data types supported by MATLAB. For each data type there is a specific set of functions that you can use for data manipulation. The first example discusses the simple case of doubling a scalar. After that, the examples discuss how to pass in, manipulate, and pass back various data types, and how to handle multiple inputs and outputs. Finally, the sections discuss passing and manipulating various MATLAB types.

Source code for the examples in this section are located in the *matlabroot*/extern/examples/refbook directory. To build these examples, make sure you have a C compiler selected using the mex -setup command. Then at the MATLAB command prompt, type:

```
mex filename.c
```

where *filename* is the name of the example.

This section looks at source code for the examples. Unless otherwise specified, the term "MEX-file" refers to a source file.

A First Example — Passing a Scalar

Let's look at a simple example of C code and its MEX-file equivalent. Here is a C computational function that takes a scalar and doubles it:

```c
#include <math.h>
void timestwo(double y[], double x[])
{
  y[0] = 2.0*x[0];
  return;
}
```

To see the same function written in the MEX-file format (`timestwo.c`), open the file in the MATLAB Editor.

In C, function argument checking is done at compile time. In MATLAB, you can pass any number or type of arguments to your M-function, which is responsible for argument checking. This is also true for MEX-files. Your program must safely handle any number of input or output arguments of any supported type.

To compile and link this example, at the MATLAB prompt, type:

```
mex timestwo.c
```

This carries out the necessary steps to create the binary MEX-file called `timestwo` with an extension corresponding to the platform on which you're running. You can now call `timestwo` as if it were an M-function:

```
x = 2;
y = timestwo(x)
y =
      4
```

You can create and compile MEX-files in MATLAB or at your operating system's prompt. MATLAB uses `mex.m`, an M-file version of the `mex` script,

and your operating system uses `mex.bat` on Microsoft Windows systems and `mex.sh` on UNIX[10] systems. In either case, typing:

```
mex filename
```

at the prompt produces a compiled version of your MEX-file.

In the above example, scalars are viewed as 1-by-1 matrices. Alternatively, you can use a special API function called `mxGetScalar` that returns the values of scalars instead of pointers to copies of scalar variables (`timestwoalt.c`). To see the alternative code (error checking has been omitted for brevity), open the file in MATLAB Editor.

This example passes the input scalar `x` by value into the `timestwo_alt` subroutine, but passes the output scalar `y` by reference.

Passing Strings

Any MATLAB type can be passed to and from MEX-files. The example `revord.c` accepts a string and returns the characters in reverse order. To see the example, open the file in MATLAB Editor.

In this example, the API function `mxCalloc` replaces `calloc`, the standard C function for dynamic memory allocation. `mxCalloc` allocates dynamic memory using the MATLAB memory manager and initializes it to zero. You must use `mxCalloc` in any situation where C would require the use of `calloc`. The same is true for `mxMalloc` and `mxRealloc`; use `mxMalloc` in any situation where C would require the use of `malloc` and use `mxRealloc` where C would require `realloc`.

Note MATLAB automatically frees up memory allocated with the `mx` allocation routines (`mxCalloc`, `mxMalloc`, `mxRealloc`) upon exiting your MEX-file. If you don't want this to happen, use the API function `mexMakeMemoryPersistent`.

---

10. UNIX is a registered trademark of The Open Group in the United States and other countries.

The gateway routine `mexFunction` allocates memory for the input and output strings. Since these are C strings, they need to be one greater than the number of elements in the MATLAB string. Next the MATLAB string is copied to the input string. Both the input and output strings are passed to the computational subroutine (`revord`), which loads the output in reverse order. Note that the output buffer is a valid null-terminated C string because `mxCalloc` initializes the memory to 0. The API function `mxCreateString` then creates a MATLAB string from the C string, `output_buf`. Finally, `plhs[0]`, the left-hand side return argument to MATLAB, is set to the MATLAB array you just created.

By isolating variables of type `mxArray` from the computational subroutine, you can avoid having to make significant changes to your original C code.

To build this example, at the command prompt type:

```
mex revord.c
```

Type:

```
x = 'hello world';
y = revord(x)
```

MATLAB displays:

```
The string to convert is 'hello world'.
y =
dlrow olleh
```

Passing Two or More Inputs or Outputs

The `plhs[]` and `prhs[]` parameters are vectors that contain pointers to each left-hand side (output) variable and each right-hand side (input) variable, respectively. Accordingly, `plhs[0]` contains a pointer to the first left-hand side argument, `plhs[1]` contains a pointer to the second left-hand side argument, and so on. Likewise, `prhs[0]` contains a pointer to the first right-hand side argument, `prhs[1]` points to the second, and so on.

This example, `xtimesy`, multiplies an input scalar by an input scalar or matrix and outputs a matrix.

To build this example, at the command prompt type:

```
mex xtimesy.c
```

Using `xtimesy` with two scalars gives:

```
x = 7;
y = 7;
z = xtimesy(x,y)

z =
     49
```

Using `xtimesy` with a scalar and a matrix gives:

```
x = 9;
y = ones(3);
z = xtimesy(x,y)

z =
     9      9      9
     9      9      9
     9      9      9
```

To see the corresponding MEX-file C code `xtimesy.c`, open the file in the MATLAB Editor.

As this example shows, creating MEX-file gateways that handle multiple inputs and outputs is straightforward. All you need to do is keep track of which indices of the vectors `prhs` and `plhs` correspond to the input and output arguments of your function. In the example above, the input variable `x` corresponds to `prhs[0]` and the input variable `y` to `prhs[1]`.

Note that `mxGetScalar` returns the value of `x` rather than a pointer to `x`. This is just an alternative way of handling scalars. You could treat `x` as a 1-by-1 matrix and use `mxGetPr` to return a pointer to `x`.

Passing Structures and Cell Arrays

Passing structures and cell arrays into MEX-files is just like passing any other data types, except the data itself is of type `mxArray`. In practice, this means that `mxGetField` (for structures) and `mxGetCell` (for cell arrays) return

pointers of type `mxArray`. You can then treat the pointers like any other pointers of type `mxArray`, but if you want to pass the data contained in the `mxArray` to a C routine, you must use an API function such as `mxGetData` to access it.

This example takes an m-by-n structure matrix as input and returns a new 1-by-1 structure that contains these fields:

- String input generates an m-by-n cell array
- Numeric input (noncomplex, scalar values) generates an m-by-n vector of numbers with the same class ID as the input, for example, `int`, `double`, and so on.

To see the program `phonebook.c`, open the file in MATLAB Editor.

To build this example, at the command prompt type:

```
mex phonebook.c
```

To see how this program works, enter this structure:

```
friends(1).name = 'Jordan Robert';
friends(1).phone = 3386;
friends(2).name = 'Mary Smith';
friends(2).phone = 3912;
friends(3).name = 'Stacy Flora';
friends(3).phone = 3238;
friends(4).name = 'Harry Alpert';
friends(4).phone = 3077;
```

The results of this input are:

```
phonebook(friends)

ans =
     name: {1x4 cell  }
    phone: [3386 3912 3238 3077]
```

Prompting User for Input

Because MATLAB does not use stdin and stdout, C functions like scanf and printf cannot be used to prompt users for input. The following example shows how to use mexCallMATLAB with the input function to get a number from the user.

```
#include "mex.h"
#include "string.h"
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
  mxArray    *new_number, *str;
  double out;

  str = mxCreateString("Enter extension:  ");
  mexCallMATLAB(1,&new_number,1,&str,"input");
  out = mxGetScalar(new_number);
  mexPrintf("You entered: %.0f ", out);
  mxDestroyArray(new_number);
  mxDestroyArray(str);
  return;
}
```

Handling Complex Data

Complex data from MATLAB is separated into real and imaginary parts. The MATLAB API provides two functions, mxGetPr and mxGetPi, that return pointers (of type double *) to the real and imaginary parts of your data.

This example, convec.c, takes two complex row vectors and convolves them. To see the example, open the file in MATLAB Editor.

To build this example, at the command prompt type:

```
mex convec.c
```

Entering these numbers at the MATLAB prompt:

```
x = [3.000 - 1.000i, 4.000 + 2.000i, 7.000 - 3.000i];
y = [8.000 - 6.000i, 12.000 + 16.000i, 40.000 - 42.000i];
```

and invoking the new MEX-file:

```
z = convec(x,y)
```

results in:

```
z =
   1.0e+02 *

Columns 1 through 4

0.1800 - 0.2600i 0.9600 + 0.2800i 1.3200 - 1.4400i 3.7600 - 0.1200i

Column 5

1.5400 - 4.1400i
```

which agrees with the results that the built-in MATLAB function `conv.m` produces.

Handling 8-,16-, and 32-Bit Data

You can create and manipulate signed and unsigned 8-, 16-, and 32-bit data from within your MEX-files. The MATLAB API provides a set of functions that support these data types. The API function `mxCreateNumericArray` constructs an unpopulated N-dimensional numeric array with a specified data size. Refer to the entry for `mxClassID` in the online reference pages for a discussion of how the MATLAB API represents these data types.

Once you have created an unpopulated MATLAB array of a specified data type, you can access the data using `mxGetData` and `mxGetImagData`. These two functions return pointers to the real and imaginary data. You can perform arithmetic on data of 8-, 16- or 32-bit precision in MEX-files and return the result to MATLAB, which will recognize the correct data class.

The example, `doubleelement.c`, constructs a 2-by-2 matrix with unsigned 16-bit integers, doubles each element, and returns both matrices to MATLAB. To see the example, open the file in MATLAB Editor.

To build this example, at the command prompt type:

```
mex doubleelement.c
```

At the MATLAB prompt, entering:

```
doubleelement
```

produces:

```
ans =
     2     6
     4     8
```

The output of this function is a 2-by-2 matrix populated with unsigned 16-bit integers.

Manipulating Multidimensional Numerical Arrays

You can manipulate multidimensional numerical arrays by using `mxGetData` and `mxGetImagData` to return pointers to the real and imaginary parts of the data stored in the original multidimensional array. The example, `findnz.c`, takes an N-dimensional array of doubles and returns the indices for the nonzero elements in the array. To see the example, open the file in the MATLAB Editor.

To build this example, at the command prompt type:

```
mex findnz.c
```

Entering a sample matrix at the MATLAB prompt gives:

```
matrix = [ 3 0 9 0; 0 8 2 4; 0 9 2 4; 3 0 9 3; 9 9 2 0]
matrix =
     3     0     9     0
     0     8     2     4
     0     9     2     4
     3     0     9     3
     9     9     2     0
```

This example determines the position of all nonzero elements in the matrix. Running the MEX-file on this matrix produces:

```
nz = findnz(matrix)
nz =
     1     1
     4     1
     5     1
     2     2
     3     2
     5     2
     1     3
     2     3
     3     3
     4     3
     5     3
     2     4
     3     4
     4     4
```

Handling Sparse Arrays

The MATLAB API provides a set of functions that allow you to create and manipulate sparse arrays from within your MEX-files. These API routines access and manipulate ir and jc, two of the parameters associated with sparse arrays. For more information on how MATLAB stores sparse arrays, see "The MATLAB Array" on page 3-17.

The example, fulltosparse.c, illustrates how to populate a sparse matrix. To see the example, open the file in MATLAB Editor.

To build this example, at the command prompt type:

```
mex fulltosparse.c
```

At the MATLAB prompt, entering:

```
full = eye(5)
full =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
```

```
     0     0     0     0     1
```

creates a full, 5-by-5 identity matrix. Using `fulltosparse` on the full matrix produces the corresponding sparse matrix.

```
spar = fulltosparse(full)
spar =
    (1,1)        1
    (2,2)        1
    (3,3)        1
    (4,4)        1
    (5,5)        1
```

Calling Functions from C MEX-Files

It is possible to call MATLAB functions, operators, M-files, and other binary MEX-files from within your C source code by using the API function `mexCallMATLAB`. The example, `sincall.c`, creates an `mxArray`, passes various pointers to a subfunction to acquire data, and calls `mexCallMATLAB` to calculate the `sine` function and plot the results. To see the example, open the file in MATLAB Editor.

To build this example, at the command prompt type:

```
mex sincall.c
```

Running this example:

```
sincall
```

displays the results:
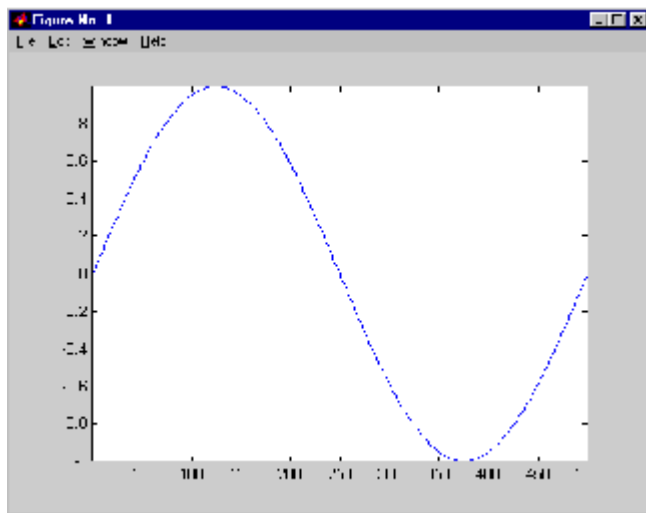
Note It is possible to generate an object of type mxUNKNOWN_CLASS using mexCallMATLAB. See the example below.

The following example creates an M-file that returns two variables but only assigns one of them a value:

```
function [a,b] = foo[c]
a = 2*c;
```

MATLAB displays the following warning message:

```
Warning: One or more output arguments not assigned during call to
'foo'.
```

If you then call foo using mexCallMATLAB, the unassigned output variable is now type mxUNKNOWN_CLASS.

Using C++ Features in MEX-Files

This example, mexcpp.cpp, illustrates how to use C++ code with your C language MEX-file. It makes use of member functions, constructors,

destructors, and the `iostream` include file. To see the example, open the file in the MATLAB Editor.

To build this example, at the command prompt type:

```
mex mexcpp.cpp
```

The calling syntax is `mexcpp(num1, num2)`.

The routine defines a class, `MyData`, with member functions `display` and `set_data`, and variables `v1` and `v2`. It constructs an object `d` of class `MyData` and displays the initialized values of `v1` and `v2`. It then sets `v1` and `v2` to your input, `num1` and `num2`, and displays the new values. Finally, cleanup of the object is done using the `delete` operator.

File Handling with C++

This example, `mexatexit.cpp`, illustrates C++ file handling features. To see the C++ code, open the C++ file in MATLAB Editor. To compare it with a C code example `mexatexit.c`, open the C file in the MATLAB Editor.

C Example

The C code example registers the `mexAtExit` function to perform cleanup tasks (close the data file) when the MEX-file clears. This example prints a message on the screen (using `mexPrintf`) when performing file operations `fopen`, `fprintf`, and `fclose`.

To build the `mexatexit.c` MEX-file, type:

```
mex mexatexit.c
```

If you type:

```
x = 'my input string';
mexatexit(x)
```

MATLAB displays:

```
Opening file matlab.data.
Writing data to file.
```

4-23

To clear the MEX-file, type:

```
clear mexatexit
```

MATLAB displays:

```
Closing file matlab.data.
```

You can see the contents of `matlab.data` by typing:

```
type matlab.data
```

MATLAB displays:

```
my input string
```

C++ Example

The C++ example does not use the `mexAtExit` function. The file open and
close functions are handled in a `fileresource` class. The destructor for this
class (which closes the data file) is automatically called when the MEX-file
clears. This example also prints a message on the screen when performing
operations on the data file. However, in this case, the only C file operation
performed is the write operation, `fprintf`.

To build the `mexatexit.cpp` MEX-file, make sure you have selected a C++
compiler, then type:

```
mex mexatexit.cpp
```

If you type:

```
z = 'for the C++ MEX-file';
mexatexit(x)
mexatexit(z)
clear mexatexit
```

MATLAB displays:

```
Writing data to file.
Writing data to file.
```

To see the contents of `matlab.data`, type:

```
type matlab.data
```

MATLAB displays:

```
my input string
for the C++ MEX-file
```

Advanced Topics

In this section...
"Help Files" on page 4-26
"Linking Multiple Files" on page 4-26
"Workspace for MEX-File Functions" on page 4-27
"Handling Large mxArrays" on page 4-27
"Memory Management" on page 4-30
"Large File I/O" on page 4-33
"Using LAPACK and BLAS Functions" on page 4-39

Help Files

Because the MATLAB interpreter chooses the binary MEX-file when both an M-file and a MEX-file with the same name are encountered in the same directory, it is possible to use M-files for documenting the behavior of your binary MEX-files. The `help` command automatically finds and displays the appropriate M-file when help is requested and the interpreter finds and executes the corresponding binary MEX-file when the function is invoked.

Linking Multiple Files

You can combine multiple source files, object files, and file libraries to build a binary MEX-file. To do this, list the additional files, with their file extensions, separated by spaces. The name of the resulting MEX-file is the name of the first file in the list.

The following command combines multiple files of different types into a binary MEX-file called `circle.ext`, where *ext* is the extension corresponding to the current platform:

```
mex circle.c square.obj rectangle.c shapes.lib
```

You may find it useful to use a software development tool like MAKE to manage MEX-file projects involving multiple source files. Simply create a MAKEFILE that contains a rule for producing object files from each of your source files,

and then invoke the `mex` build script to combine your object files into a binary MEX-file. This way you can ensure that your source files are recompiled only when necessary.

Workspace for MEX-File Functions

Unlike M-file functions, MEX-file functions (binary MEX-files) do not have their own variable workspace. MEX-file functions operate in the caller's workspace. `mexEvalString` evaluates the string in the caller's workspace. In addition, you can use the `mexGetVariable` and `mexPutVariable` routines to get and put variables into the caller's workspace.

Handling Large mxArrays

Binary MEX-files built on 64-bit platforms can handle 64-bit `mxArrays`. These large data arrays can have up to $2^{48}-1$ elements. The maximum number of elements a sparse `mxArray` can have is $2^{48}-2$.

Using the following instructions creates platform-independent binary MEX-files as well.

Your system configuration can impact the performance of MATLAB. The 64-bit processor requirement enables you to create the `mxArray` and access data in it. However, your system's memory, in particular the size of RAM and virtual memory, determine the speed at which MATLAB processes the `mxArray`. The more memory available, the faster the processing.

The amount of RAM also limits the amount of data you can process at one time in MATLAB. For guidance on memory issues, see "Strategies for Efficient Use of Memory" in the Programming Fundamentals documentation. Memory management within source MEX-files can have special considerations, as described in "Memory Management" on page 4-30.

Using the 64-Bit API

To work with a 64-bit `mxArray`, your source code must comply with the 64-bit API, which consists of the functions in the following table.

mxCalcSingleSubscript	mxCreateSparseLogicalMatrix
mxCalloc	mxCreateStructArray
mxCopyCharacterToPtr	mxCreateStructMatrix
mxCopyComplex16ToPtr	mxGetCell
mxCopyComplex8ToPtr	mxGetDimensions
mxCopyInteger1ToPtr	mxGetElementSize
mxCopyInteger2ToPtr	mxGetField
mxCopyInteger4ToPtr	mxGetFieldByNumber
mxCopyPtrToCharacter	mxGetIr
mxCopyPtrToComplex16	mxGetJc
mxCopyPtrToComplex8	mxGetM
mxCopyPtrToInteger1	mxGetN
mxCopyPtrToInteger2	mxGetNumberOfDimensions
mxCopyPtrToInteger4	mxGetNumberOfElements
mxCopyPtrToPtrArray	mxGetNzmax
mxCopyPtrToReal4	mxGetProperty
mxCopyPtrToReal8	mxGetString
mxCopyReal4ToPtr	mxMalloc
mxCopyReal8ToPtr	mxRealloc
mxCreateCellArray	mxSetCell
mxCreateCellMatrix	mxSetDimensions
mxCreateCharArray	mxSetField
mxCreateCharMatrixFromStrings	mxSetFieldByNumber
mxCreateDoubleMatrix	mxSetIr
mxCreateLogicalArray	mxSetJc
mxCreateLogicalMatrix	mxSetM
mxCreateNumericArray	mxSetN

```
mxCreateNumericMatrix            mxSetNzmax

mxCreateSparse                   mxSetProperty
```

Functions in this API use the `mwIndex` and `mwSize` types. For information about using these macros, see "Required Header Files" on page 4-4.

Building the Binary MEX-File

Use the `mex` build script option `-largeArrayDims` with the 64-bit API.

Example

The example, `arraySize.c` in *matlabroot*/extern/examples/mex, illustrates memory requirements of large `mxArrays`. To see the example, open the file in MATLAB Editor.

This function requires one positive scalar numeric input, which it uses to create a square matrix. It checks the size of the input to make sure your system can theoretically create a matrix of this size. If the input is valid, it displays the size of the mxArray in kilobytes.

To build this MEX-file, type:

```
mex -largeArrayDims arraySize.c
```

To run the MEX-file, type:

```
arraySize(2^10)
```

If your system has enough available memory, MATLAB displays:

```
Dimensions: 1024 x 1024
Size of array in kilobytes: 1024
```

If your system does not have enough memory to create the array, MATLAB displays an `Out of memory` error.

You can experiment with this function to test the performance and limits of handling large arrays on your system.

Caution Using Negative Values

When using the 64-bit API, `mwSize` and `mwIndex` are equivalent to `size_t` in C or `INTEGER*8` in Fortran. These types are unsigned, unlike `int` and `INTEGER*4`, which are the types used in the 32-bit API. Be careful not to pass any negative values to functions that take `mwSize` or `mwIndex` arguments. Do not cast negative `int` or `INTEGER*4` values to `mwSize` or `mwIndex`; the returned value can not be predicted. Instead, change your code to avoid using negative values.

Building Cross-Platform Applications

If you develop cross-platform applications (programs that can run on both 32- and 64-bit architectures), you must pay attention to the upper limit of values you use for `mwSize` and `mwIndex`. The 32-bit application reads these values and assigns them to variables declared as `int` in C or `INTEGER*4` in Fortran. Be careful to avoid assigning a large `mwSize` or `mwIndex` value to an `int`, `INTEGER*4`, or other variable that might be too small.

Memory Management

Memory management in MEX-files is similar to memory management in any C or Fortran application. However, there are special considerations because a binary MEX-file exists within the context of a larger application, i.e., MATLAB.

- "Automatic Cleanup of Temporary Arrays" on page 4-30

- "Persistent Arrays" on page 4-31

- "Hybrid Arrays" on page 4-32

To avoid common problems related to memory management, see "Memory Management Issues" on page 3-55.

Automatic Cleanup of Temporary Arrays

When a binary MEX-file returns control to MATLAB, it returns the results of its computations in the output arguments—the `mxArrays` contained in the left-hand side arguments `plhs[]`. MATLAB destroys any `mxArray` created by the MEX-file that is not in this argument list. In addition, MATLAB frees any

memory that was allocated in the MEX-file using the `mxCalloc`, `mxMalloc`, or `mxRealloc` functions.

In general, we recommend that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism.However, there are several circumstances in which the MEX-file does not reach its normal return statement.

The normal return is not reached if:

- A call to `mexErrMsgTxt` occurs.

- A call to `mexCallMATLAB` occurs and the function being called creates an error. (A source MEX-file can trap such errors by using the `mexSetTrapFlag` function, but not all MEX-files necessarily need to trap errors.)

- The user interrupts the binary MEX-file's execution using **Ctrl+C**.

- The binary MEX-file runs out of memory. When this happens, the MATLAB out-of-memory handler immediately terminates the MEX-file.

A careful MEX-file programmer can ensure safe cleanup of all temporary arrays and memory before returning in the first two cases, but not in the last two cases. In the last two cases, the automatic cleanup mechanism is necessary to prevent memory leaks.

Persistent Arrays

You can exempt an array, or a piece of memory, from the MATLAB automatic cleanup by calling `mexMakeArrayPersistent` or `mexMakeMemoryPersistent`. However, if a binary MEX-file creates such persistent objects, there is a danger that a memory leak could occur if the MEX-file is cleared before the persistent object is properly destroyed. To prevent this from happening, a source MEX-file that creates persistent objects should register a function, using the `mexAtExit` function, which disposes of the objects. (You can use a `mexAtExit` function to dispose of other resources as well; for example, you can use `mexAtExit` to close an open file.)

For example, here is a simple source MEX-file that creates a persistent array and properly disposes of it.

```
#include "mex.h"

static int initialized = 0;
static mxArray *persistent_array_ptr = NULL;

void cleanup(void) {
    mexPrintf("MEX-file is terminating, destroying array\n");
    mxDestroyArray(persistent_array_ptr);
}

void mexFunction(int nlhs,
    mxArray *plhs[],
    int nrhs,
    const mxArray *prhs[])
{
  if (!initialized) {
    mexPrintf("MEX-file initializing, creating array\n");

    /* Create persistent array and register its cleanup. */
    persistent_array_ptr = mxCreateDoubleMatrix(1, 1, mxREAL);
    mexMakeArrayPersistent(persistent_array_ptr);
    mexAtExit(cleanup);
    initialized = 1;

    /* Set the data of the array to some interesting value. */
    *mxGetPr(persistent_array_ptr) = 1.0;
  } else {
    mexPrintf("MEX-file executing; value of first array
        element is %g\n", *mxGetPr(persistent_array_ptr));
  }
}
```

Hybrid Arrays

Functions such as mxSetPr, mxSetData, and mxSetCell allow the direct placement of memory pieces into an mxArray. mxDestroyArray destroys these pieces along with the entire array. Because of this, it is possible to create an array that cannot be destroyed, i.e., an array on which it is not safe to call mxDestroyArray. Such an array is called a *hybrid* array, because it contains both destroyable and nondestroyable components.

For example, it is not legal to call `mxFree` (or the ANSI `free()` function, for that matter) on automatic variables. Therefore, in the following code fragment, `pArray` is a hybrid array.

```
mxArray *pArray = mxCreateDoubleMatrix(0, 0, mxREAL);
double data[10];

mxSetPr(pArray, data);
mxSetM(pArray, 1);
mxSetN(pArray, 10);
```

Another example of a hybrid array is a cell array or structure, one of whose children is a read-only array (an array with the `const` qualifier, such as one of the inputs to the MEX-file). The array cannot be destroyed because the input to the MEX-file would also be destroyed.

Because hybrid arrays cannot be destroyed, they cannot be cleaned up by the automatic mechanism outlined in "Automatic Cleanup of Temporary Arrays" on page 4-30. As described in that section, the automatic cleanup mechanism is the only way to destroy temporary arrays in case of a user interrupt. Therefore, *temporary hybrid arrays are illegal* and can cause your binary MEX-file to crash. Although persistent hybrid arrays are viable, it is best to avoid using them whenever possible.

Large File I/O

MATLAB supports the use of 64-bit file I/O operations in your MEX-file programs. This enables you to read and write data to files that are up to and greater than 2 GB (2^{31-1} bytes) in size. Note that some operating systems or compilers might not support files larger than 2 GB.

This section covers the following topics on large file I/O:

- "Prerequisites to Using 64-Bit I/O" on page 4-34

- "Specifying Constant Literal Values" on page 4-36

- "Opening a File" on page 4-36

- "Printing Formatted Messages" on page 4-37

- "Replacing fseek and ftell with 64-Bit Functions" on page 4-37

Prerequisites to Using 64-Bit I/O

This section describes the components you need to use 64-bit file I/O in your MEX-file programs:

Header File. Header file `io64.h` defines many of the types and functions required for 64-bit file I/O. The statement to include this file must be the *first* `#include` statement in your source file and must also precede any system header include statements:

```
#include "io64.h"
#include "mex.h"
        .
        .
        .
```

Type Declarations. Use the following types to declare variables used in 64-bit file I/O.

MEX Type	Description	POSIX
fpos_T	Declares a 64-bit int type for setFilePos() and getFilePos(). Defined in io64.h.	fpos_t
int64_T, uint64_T	Declares 64-bit signed and unsigned integer types. Defined in tmwtypes.h.	long, long
structStat	Declares a structure to hold the size of a file. Defined in io64.h.	struct stat

MEX Type	Description	POSIX
FMT64	Used in `mexPrintf` to specify length within a format specifier such as `%d`. See example in the section "Printing Formatted Messages" on page 4-37. `FMT64` is defined in `tmwtypes.h`.	`%lld`
LL, LLU	Suffixes for literal `int` constant 64-bit values (C Standard ISO/IEC 9899:1999(E) Section 6.4.4.1). Used only on UNIX[11] systems.	LL, LLU

Functions. Here are the functions you need for 64-bit file I/O. All are defined in the header file `io64.h`.

Function	Description	POSIX
fileno()	Gets a file descriptor from a file pointer	fileno()
fopen()	Opens the file and obtains the file pointer	fopen()
getFileFstat()	Gets the file size of a given file pointer	fstat()
getFilePos()	Gets the file position for the next I/O	fgetpos()
getFileStat()	Gets the file size of a given filename	stat()
setFilePos()	Sets the file position for the next I/O	fsetpos()

---

11. UNIX is a registered trademark of The Open Group in the United States and other countries.

Specifying Constant Literal Values

To assign signed and unsigned 64-bit integer literal values, use type definitions int64_T and uint64_T.

On UNIX systems, to assign a literal value to an integer variable where the value to be assigned is greater than $2^{31}-1$ signed, you must suffix the value with LL. If the value is greater than $2^{32}-1$ unsigned, then use LLU as the suffix. These suffixes apply only to UNIX systems and are considered invalid on the Microsoft Windows systems.

> **Note** The LL and LLU suffixes are not required for hardcoded (literal) values less than 2 G ($2^{31}-1$), even if they are assigned to a 64-bit int type.

The following example declares a 64-bit integer variable initialized with a large literal int value, and two 64-bit integer variables:

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
                 const mxArray *prhs[])
{
#if defined(_MSC_VER) || defined(__BORLANDC__)      /* Windows */
   int64_T large_offset_example = 9000222000;
#else                                               /* UNIX    */
   int64_T large_offset_example = 9000222000LL;
#endif

int64_T offset   = 0;
int64_T position = 0;
```

Opening a File

To open a file for reading or writing, use the C fopen function as you normally would. As long as you have included io64.h at the start of your program, fopen works correctly for large files. No changes at all are required for fread, fwrite, fprintf, fscanf, and fclose.

To open an existing file for read and update in binary mode:

```
fp = fopen(filename, "r+b");
if (NULL == fp)
    {
    /* File does not exist. Create new file for writing
     * in binary mode.
     */
    fp = fopen(filename, "wb");
    if (NULL == fp)
        {
        sprintf(str, "Failed to open/create test file '%s'",
                filename);
        mexErrMsgTxt(str);
        return;
        }
    else
        {
        mexPrintf("New test file '%s' created\n",filename);
        }
    }
else mexPrintf("Existing test file '%s' opened\n",filename);
```

Printing Formatted Messages

You cannot print 64-bit integers using the `%d` conversion specifier. Instead, use `FMT64` to specify the appropriate format for your platform. `FMT64` is defined in the header file `tmwtypes.h`. The following example shows how to print a message showing the size of a large file:

```
int64_T large_offset_example = 9000222000LL;

mexPrintf("Example large file size: %" FMT64 "d bytes.\n",
            large_offset_example);
```

Replacing fseek and ftell with 64-Bit Functions

The ANSI C `fseek` and `ftell` functions are not 64-bit file I/O capable on most platforms. The functions `setFilePos` and `getFilePos`, however, are defined as the corresponding POSIX `fsetpos` and `fgetpos`, (or `fsetpos64` and

fgetpos64), as required by your platform/OS. These functions are 64-bit file I/O capable on all platforms.

The following example shows how to use setFilePos instead of fseek, and getFilePos instead of ftell. It uses getFileFstat to find the size of the file, and then uses setFilePos to seek to the end of the file to prepare for adding data at the end of the file.

> **Note** Although the offset parameter to setFilePos and getFilePos is really a pointer to a signed 64-bit integer, int64_T, it must be cast to an fpos_T*. The fpos_T type is defined in io64.h as the appropriate fpos64_t or fpos_t, as required by your platform/OS.

```
getFileFstat(fileno(fp), &statbuf);
fileSize = statbuf.st_size;
offset = fileSize;

setFilePos(fp, (fpos_T*) &offset);
getFilePos(fp, (fpos_T*) &position );
```

Unlike fseek, setFilePos supports only absolute seeking relative to the beginning of the file. If you want to do a relative seek, first call getFileFstat to obtain the file size, and then convert the relative offset to an absolute offset that you can pass to setFilePos.

Determining the Size of an Open File

Getting the size of an open file involves two steps:

1 Refresh the record of the file size stored in memory using getFilePos and setFilePos.

2 Retrieve the size of the file using getFileFstat.

Refreshing the File Size Record. Before attempting to retrieve the size of an open file, you should first refresh the record of the file size residing in memory. If you skip this step on a file that is opened for writing, the file size returned might be incorrect or 0.

To refresh the file size record, seek to any offset in the file using `setFilePos`. If you do not want to change the position of the file pointer, you can seek to the current position in the file. This example obtains the current offset from the start of the file, and then seeks to the current position to update the file size without moving the file pointer:

```
getFilePos( fp, (fpos_T*) &position);
setFilePos( fp, (fpos_T*) &position);
```

Getting the File Size. The `getFileFstat` function takes a file descriptor input argument (that you can obtain from the file pointer of the open file using `fileno`) and returns the size of that file in bytes in the `st_size` field of a `structStat` structure:

```
structStat statbuf;
int64_T fileSize = 0;

if (0 == getFileFstat(fileno(fp), &statbuf))
    {
    fileSize = statbuf.st_size;
    mexPrintf("File size is %" FMT64 "d bytes\n", fileSize);
    }
```

Determining the Size of a Closed File

The `getFileStat` function takes the filename of a closed file as an input argument and returns the size of the file in bytes in the `st_size` field of a `structStat` structure:

```
structStat statbuf;
int64_T fileSize = 0;

if (0 == getFileStat(filename, &statbuf))
    {
    fileSize = statbuf.st_size;
    mexPrintf("File size is %" FMT64 "d bytes\n", fileSize);
    }
```

Using LAPACK and BLAS Functions

LAPACK is a large, multiauthor Fortran subroutine library that MATLAB uses for numerical linear algebra. BLAS, which stands for Basic Linear

Algebra Subroutines, is used by MATLAB to speed up matrix multiplication and the LAPACK routines themselves. The functions provided by LAPACK and BLAS can also be called directly from within your C source MEX-files.

This section explains how to write and build MEX-files that call LAPACK and BLAS functions. It provides information on

- "Specifying the Function Name" on page 4-40
- "Calling LAPACK and BLAS Functions from C" on page 4-41
- "Handling Complex Numbers" on page 4-42
- "Preserving Input Values from Modification" on page 4-43
- "Building the C MEX-File" on page 4-44
- "Example — Symmetric Indefinite Factorization Using LAPACK" on page 4-46
- "Calling LAPACK and BLAS Functions from Fortran" on page 4-46
- "Building the Fortran MEX-File" on page 4-47

Source code for the examples in this section are located in the *matlabroot*/extern/examples/refbook directory. To build these examples, make sure you have the appropriate C or Fortran compiler selected using the mex -setup command. Then at the MATLAB command prompt, type:

```
mex filename.ext
```

where *filename.ext* is the name of the source code for the example.

Specifying the Function Name

When calling an LAPACK or BLAS function, some platforms require an underscore character following the function name in the call statement.

On the Microsoft Windows platform use the function name alone, with no trailing underscore. For example, to call the LAPACK dgemm function, use:

```
dgemm(arg1, arg2, ..., argn);
```

On the Sun Solaris, Linus Torvalds' Linux, and AppleMacintosh platforms, add the underscore after the function name. For example, to call dgemm on any of these platforms, use:

```
dgemm_(arg1, arg2, ..., argn);
```

Calling LAPACK and BLAS Functions from C

Since the LAPACK and BLAS functions are written in Fortran, arguments passed to and from these functions must be passed by reference. The following example calls dgemm, passing all arguments by reference. An ampersand (&) precedes each argument unless that argument is already a reference.

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  double *A, *B, *C, one = 1.0, zero = 0.0;
  mwSize m,n,p;
  char *chn = "N";

  A = mxGetPr(prhs[0]);
  B = mxGetPr(prhs[1]);
  m = mxGetM(prhs[0]);
  p = mxGetN(prhs[0]);
  n = mxGetN(prhs[1]);

  if (p != mxGetM(prhs[1])) {mexErrMsgTxt
    ("Inner dimensions of matrix multiply do not match");
  }

  plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
  C = mxGetPr(plhs[0]);

  /* Pass all arguments to Fortran by reference */
  dgemm(chn, chn, &m, &n, &p, &one, A, &m, B, &p, &zero, C, &m);
}
```

Handling Complex Numbers

Caution Use care when using level 1 BLAS functions (e.g. ZDOTU, ZDOTC) with complex numbers in a C program MEX-file.

MATLAB stores complex numbers differently than Fortran. MATLAB stores the real and imaginary parts of a complex number in separate, equal length vectors, pr and pi. Fortran stores the same number in one location with the real and imaginary parts interleaved.

As a result, complex variables exchanged between MATLAB and the Fortran functions in LAPACK and BLAS are incompatible. MATLAB provides conversion routines that change the storage format of complex numbers to address this incompatibility.

Input Arguments. For all complex variables passed as input arguments to a Fortran function, you need to convert the storage of the MATLAB variable to be compatible with the Fortran function. Use the mat2fort function for this. See the example that follows.

Output Arguments. For all complex variables passed as output arguments to a Fortran function, you need to do the following:

1 When allocating storage for the complex variable, allocate a real variable with twice as much space as you would for a MATLAB variable of the same size. You need to do this because the returned variable uses the Fortran format, which takes twice the space. See the allocation of zout in the example that follows.

2 Once the variable is returned to MATLAB, convert its storage so that it is compatible with MATLAB. Use the fort2mat function for this.

Example — Passing Complex Variables. The example below shows how to call an LAPACK function from MATLAB, passing complex prhs[0] as input and receiving complex plhs[0] as output. Temporary variables zin and zout are used to hold prhs[0] and plhs[0] in Fortran format.

```
#include "mex.h"
```

```
#include "fort.h"       /* defines mat2fort and fort2mat */

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray
*prhs[])
{
  mwSize lda, n;
  double *zin, *zout;
  lda = mxGetM(prhs[0]);
  n = mxGetN(prhs[0]);

  /* Convert input to Fortran format */
  zin = mat2fort(prhs[0], lda, n);

  /* Allocate a real, complex, lda-by-n variable to store output
*/
  zout = mxCalloc(2*lda*n, sizeof(double));

  /* Call complex LAPACK function */
  zlapack_function(zin, &lda, &n, zout);

  /* Convert output to MATLAB format */
  plhs[0] = fort2mat(zout, lda, lda, n);

  /* Free intermediate Fortran format arrays */
  mxFree(zin);
  mxFree(zout);
}
```

Preserving Input Values from Modification

Many LAPACK and BLAS functions modify the values of arguments passed
in to them. It is advisable to make a copy of arguments that can be modified
prior to passing them to the function. For complex inputs, this point is moot
since the mat2fort version of the input is a new piece of memory, but for
real data this is not the case.

The following example calls an LAPACK function that modifies the first input
argument. The code in this example makes a copy of prhs[0], and then
passes the copy to the LAPACK function to preserve the contents of prhs[0].

```
/* lapack_function modifies A so make a copy of the input */
m = mxGetM(prhs[0]);
n = mxGetN(prhs[0]);
A = mxCalloc(m*n, sizeof(double));

/* Copy mxGetPr(prhs[0]) into A */
temp = mxGetPr(prhs[0]);
for (k = 0; k < m*n; k++) {
   A[k] = temp[k];
   }

/* lapack_function does not modify B
/* so it is OK to use the input
directly */
B = mxGetPr(prhs[1]);
lapack_function(A, B);        /* modifies A but not B */

/* Free A when you are done with it */
mxFree(A);
```

Building the C MEX-File

The examples in this section show how to compile and link a C source MEX file `myCmexFile.c` on the platforms supported by MATLAB.

Building on Windows Systems. If you build your C MEX-file on a Windows platform, you need to explicitly specify a library file to link with.

On a Windows system, use this command if you are using the Lcc compiler that ships with MATLAB:

```
mex myCmexFile.c
matlabroot\extern\lib\win32\lcc\libmwlapack.lib
matlabroot\extern\lib\win32\lcc\libmwblas.lib
```

Or, use this command if you are using the Microsoft Visual C++ compiler:

```
mex myCmexFile.c
matlabroot\extern\lib\win32\microsoft\libmwlapack.lib
matlabroot\extern\lib\win32\microsoft\libmwblas.lib
```

or:

```
mex myCmexFile.c
matlabroot\extern\lib\win64\microsoft\libmwlapack.lib
matlabroot\extern\lib\win64\microsoft\libmwblas.lib
```

Building on Other Operating Systems. On all other platforms, you can build your MEX-file as you would any other C source MEX-file. For example:

```
mex myCmexFile.c -lmwlapack -lmwblas
```

MEX-File Functions That Perform Complex Number Conversion.
MATLAB supplies the files `fort.c` and `fort.h`, which provide routines for conversion between MATLAB and FORTRAN complex data structures. These files define the `mat2fort` and `fort2mat` routines mentioned previously under "Handling Complex Numbers" on page 4-42.

If your program uses these routines, you need to:

1 Include the `fort.h` file in your program using `#include "fort.h"`. See "Example — Passing Complex Variables" on page 4-42.

2 Build the `fort.c` file with your program. Specify the path, *matlabroot*/extern/examples/refbook for both `fort.c` and `fort.h` in the build command.

On Windows systems, use either one of the following:

1

```
mex myCmexFile.c matlabroot/extern/examples/refbook/fort.c
-Imatlabroot/extern/examples/refbook
matlabroot/extern/lib/win32/microsoft/libmwlapack.lib
matlabroot/extern/lib/win32/microsoft/libmwblas.lib
```

2

```
mex myCmexFile.c matlabroot/extern/examples/refbook/fort.c
-Imatlabroot/extern/examples/refbook
matlabroot/extern/lib/win32/lcc/libmwlapack.lib
matlabroot/extern/lib/win32/lcc/libmwblas.lib
```

For all other platforms, use:

```
mex myCmexFile.c matlabroot/extern/examples/refbook/fort.c
-Imatlabroot/extern/examples/refbook
```

Example — Symmetric Indefinite Factorization Using LAPACK

The directory *matlabroot*/extern/examples/refbook contains an example C source MEX-file that calls two LAPACK functions. There are two versions of this file:

- utdu_slv.c - calls functions zhesvx and dsysvx, and thus is compatible with the Windows platform.

- utdu_slv_.c - calls functions zhesvx_ and dsysvx_, and thus is compatible with the Linux, Solaris, and Macintosh platforms.

Calling LAPACK and BLAS Functions from Fortran

You can make calls to the LAPACK and BLAS functions used by MATLAB from your Fortran MEX files. The following is an example program that takes two matrices and multiplies them by calling the LAPACK routine, dgemm:

```fortran
#include "fintrf.h"

subroutine mexFunction(nlhs, plhs, nrhs, prhs)
mwPointer plhs(*), prhs(*)
integer nlhs, nrhs
mwPointer mxcreatedoublematrix
mwPointer mxgetpr
mwPointer A, B, C
mwSize mxgetm, mxgetn
mwSize m, n, p, numel
double precision one, zero, ar, br
character ch1, ch2

ch1 = 'N'
ch2 = 'N'
one = 1.0
zero = 0.0
```

```
A = mxgetpr(prhs(1))
B = mxgetpr(prhs(2))
m = mxgetm(prhs(1))
p = mxgetn(prhs(1))
n = mxgetn(prhs(2))

plhs(1) = mxcreatedoublematrix(m, n, 0.0)
C = mxgetpr(plhs(1))
numel = 1
call mxcopyptrtoreal8(A, ar, numel)
call mxcopyptrtoreal8(B, br, numel)

call dgemm(ch1, ch2, m, n, p, one, %val(A), m,
+             %val(B), p, zero, %val(C), m)

return
end
```

Building the Fortran MEX-File

The examples in this section show how to compile and link a Fortran MEX file, `myFortranmexFile.F`, on the platforms supported by MATLAB.

Building on the Windows Platform. On the Windows platform, using Visual Fortran, link against the libraries `libdflapack.lib` and `libdfblas.lib`:

```
mex -v myFortranMexFile.F
matlabroot/extern/lib/win32/microsoft/libdflapack.lib
matlabroot/extern/lib/win32/microsoft/libdfblas.lib
```

Building on Other UNIX Platforms. On the UNIX platforms, create the MEX file as follows:

```
mex -v myFortranMexFile.F -lmwlapack -lmwblas
```

Debugging C Language MEX-Files

In this section...
"Notes on Debugging" on page 4-48
"Debugging on the Microsoft Windows Platforms" on page 4-48
"Debugging on Linux Platforms" on page 4-56

Notes on Debugging

The examples show how to debug yprime.c, found in your *matlabroot*/extern/examples/mex/ directory.

Binary MEX-files built with the -g option do not execute on other computers because they rely on files that are not distributed with MATLAB software. Refer to the "Calling C and Fortran Programs from MATLAB" topic "Troubleshooting MEX-Files" on page 3-47 for additional information on isolating problems with MEX-files.

Debugging on the Microsoft Windows Platforms

The Microsoft® Visual Studio® development environment provides complete source code debugging, including the ability to set breakpoints, examine variables, and step through the source code line-by-line.

For information on debugging MEX-files compiled with other MATLAB supported compilers, see Technical Note 1605, MEX-files Guide, at http://www.mathworks.com/support/tech-notes/1600/1605.html.

Visual Studio 2005

This section describes how to debug using the default compiler, that is, the compiler used to build MATLAB.

1 Select the Microsoft Visual C++ 2005 compiler. At the MATLAB prompt, type:

```
mex -setup
```

Type y to locate installed compilers, and then type the number corresponding to this compiler.

2 Next, compile the source MEX-file with the -g option, which builds the file with debugging symbols included. For example:

```
mex -g yprime.c
```

On a 32–bit platform, this command creates the executable file yprime.mexw32.

3 Start Visual Studio®. Do not exit your MATLAB session.

4 From the Visual Studio **Tools** menu, select **Attach to Process...**[12]

---
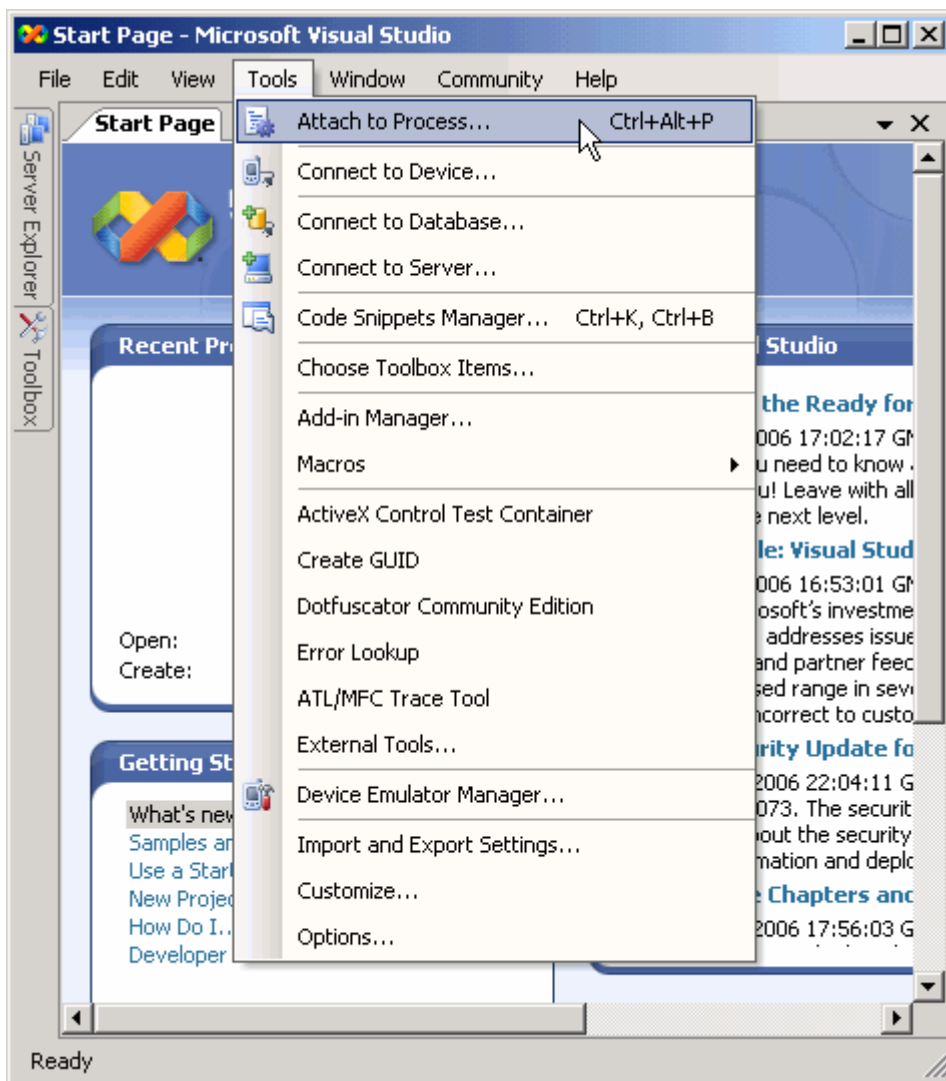
12. used by permission

5 In the Attach to Process dialog box, select the MATLAB process and click
Attach.

Attach to Process						? ☒
Transport:	Default					▼
Qualifier:	HOWELLH				▼	Browse...

Transport Information
The default transport lets you select processes on this computer or a remote computer running the Microsoft Visual Studio Remote Debugging Monitor (MSVSMON.EXE).

Attach to:	Automatic: Native code					Select...

Available Processes

Process	ID	Title	Type	User Name	Session
acrotray.exe	2496		x86	MATHWORKS\hh...	0
ctfmon.exe	2520		x86	MATHWORKS\hh...	0
explorer.exe	2224	H:\Documents\SnagIt Catalog	x86	MATHWORKS\hh...	0
MATLAB.exe	3008	MATLAB	x86	MATHWORKS\hh...	0
Realmon.exe	2512		x86	MATHWORKS\hh...	0
rundll32.exe	2448		x86	MATHWORKS\hh...	0
SnagIt32.exe	3528	SnagIt	x86	MATHWORKS\hh...	0
TSCHelp.exe	3552		x86	MATHWORKS\hh...	0

☐ Show processes from all users   ☐ Show processes in all sessions   Refresh

Attach   Cancel

Visual Studio loads data then displays an empty code pane.
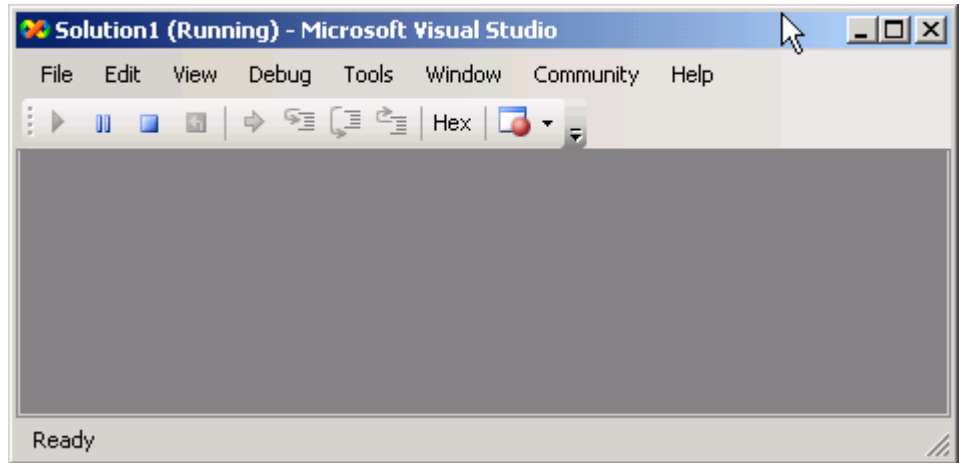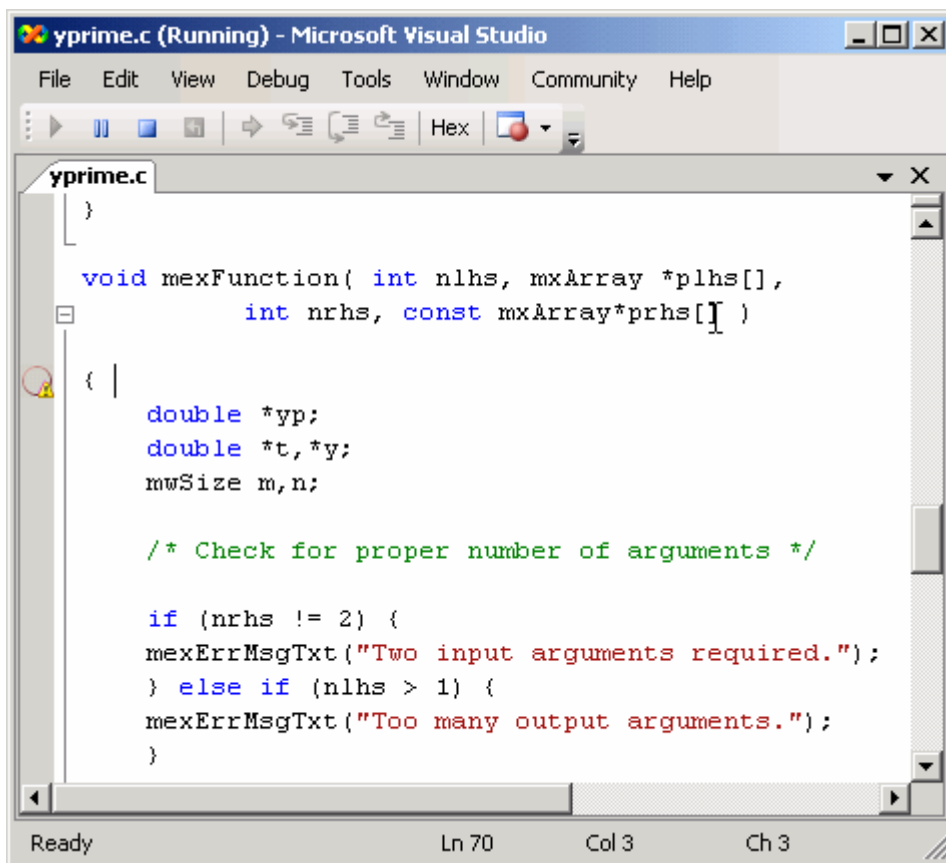
6 Open the source file yprime.c by selecting **File > Open > File**. yprime.c is found in the *matlabroot*/extern/examples/mex/ directory.
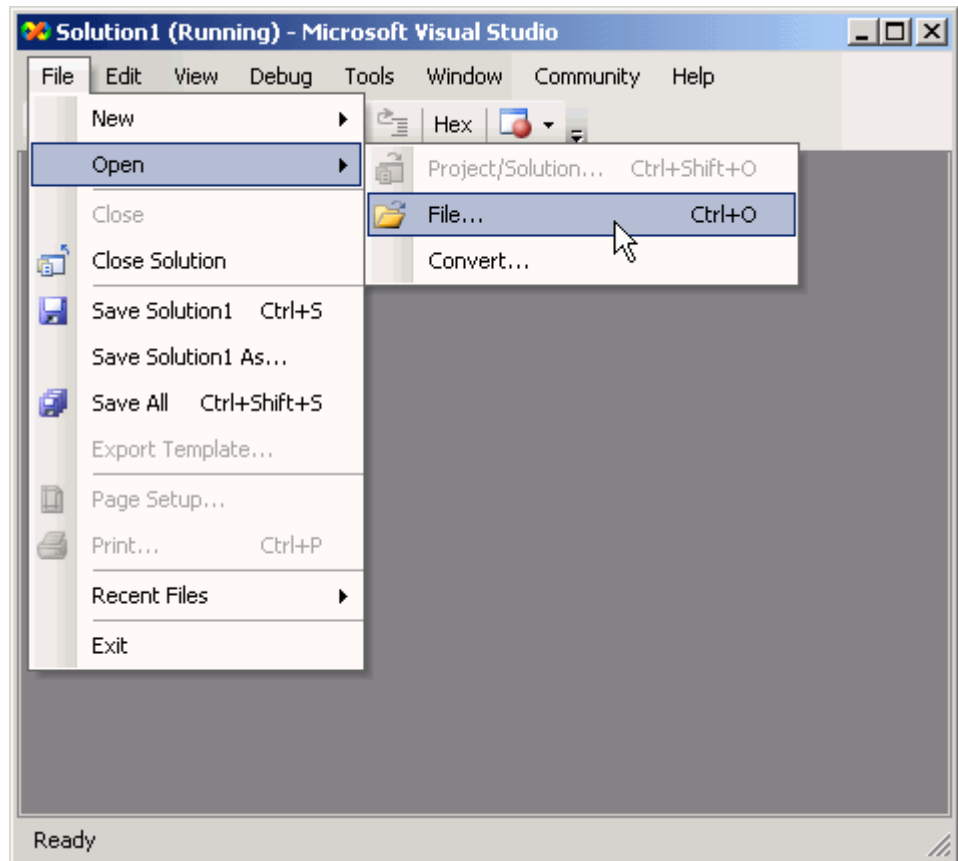
7 Set a breakpoint by right-clicking the desired line of code and following **Breakpoint > Insert Breakpoint** on the context menu. It is often

convenient to set a breakpoint at `mexFunction` to stop at the beginning of the gateway routine.

If you have not yet run the executable file, ignore any "!" icon that appears with the breakpoint next to the line of code.

Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.

8 Run the binary MEX-file in MATLAB. After typing:

```
yprime(1,1:4)
```

yprime.c is opened in the Visual Studio debugger at the first breakpoint.

9 If you select **Debug > Continue**, MATLAB displays:

```
ans =

    2.0000    8.9685    4.0000   -1.0947
```

For more information on how to debug in the Visual Studio environment, see your Microsoft documentation.

Debugging on Linux Platforms

The GNU Debugger gdb, available on Linux[13] systems, provides complete source code debugging, including the ability to set breakpoints, examine variables, and step through the source code line-by-line.

For information on debugging MEX-files compiled with other MATLAB supported compilers, see Technical Note 1605, MEX-files Guide, at `http://www.mathworks.com/support/tech-notes/1600/1605.html`.

GNU Debugger gdb

In this procedure, the MATLAB command prompt >> is shown in front of MATLAB commands, and linux> represents a Linux prompt; your system may show a different prompt. The debugger prompt is <gdb>.

To debug with gdb:

1 Compile the source MEX-file with the -g option, which builds the file with debugging symbols included. For this example, at the Linux prompt, type:

```
linux> mex -g yprime.c
```

On a Linux 32–bit platform, this command creates the executable file yprime.mexglx.

2 At the Linux prompt, start the gdb debugger using the matlab function -D option:

---

13. Linux is a registered trademark of Linus Torvalds.

```
linux> matlab -Dgdb
```

3 Start MATLAB without the Java™ Virtual Machine (JVM™) by using the -nojvm startup flag:

```
<gdb> run -nojvm
```

4 In MATLAB, enable debugging with the dbmex function and run your binary MEX-file:

```
>> dbmex on
>> yprime(1,1:4)
```

5 At this point, you are ready to start debugging.

It is often convenient to set a breakpoint at mexFunction so you stop at the beginning of the gateway routine.

```
<gdb> break mexFunction
<gdb> continue
```

6 Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.

To proceed from a breakpoint, type:

```
<gdb> continue
```

7 After stopping at the last breakpoint, type:

```
<gdb> continue
```

yprime finishes and MATLAB displays:

```
ans =

    2.0000    8.9685    4.0000   -1.0947
```

8 From the MATLAB prompt you can return control to the debugger by typing:

```
>> dbmex stop
```

Or, if you are finished running MATLAB, type:

```
>> quit
```

9 When you are finished with the debugger, type:

```
<gdb> quit
```

You return to the Linux prompt.

Refer to the documentation provided with your debugger for more information on its use.

5

Creating Fortran
MEX-Files

Fortran Source MEX-Files

In this section...
"The Components of a Fortran MEX-File" on page 5-2
"Gateway Routine" on page 5-2
"Computational Routine" on page 5-5
"Preprocessor Macros" on page 5-5
"Using the Fortran %val Construct" on page 5-6
"Data Flow in MEX-Files" on page 5-7

The Components of a Fortran MEX-File

You create binary MEX-files using the `mex` build script. `mex` compiles and links source MEX-file files into a shared library called a binary MEX-file, which you can run from the MATLAB command line. Once compiled, you treat binary MEX-files exactly like MATLAB M-files and built-in functions.

This section explains the components of a source MEX-file, statements you use in a program source file. Unless otherwise specified, the term "MEX-file" refers to a source file.

The MEX-file consists of:

- A "Gateway Routine" on page 5-2 that interfaces Fortran and MATLAB data.
- A "Computational Routine" on page 5-5 that performs the computations you want implemented in the binary MEX-file.
- "Preprocessor Macros" on page 5-5 for building platform-independent code.

Gateway Routine

The *gateway routine* is the entry point to the MEX-file shared library. It is through this routine that MATLAB accesses the rest of the routines in your MEX-files. Use the following guideline to create a gateway routine:

- "Naming the Gateway Routine" on page 5-3

A Fortran MEX-file gateway routine looks like this:

```
C      The gateway routine.
       subroutine mexFunction(nlhs, plhs, nrhs, prhs)
       integer nlhs, nrhs
       mwpointer plhs(*), prhs(*)
```

Naming the Gateway Routine

The name of the gateway routine must be `mexFunction`.

Required Parameters

A gateway routine must contain the parameters `prhs`, `nrhs`, `plhs`, and `nlhs` which are described in the following table.

Parameter	Description
prhs	An array of right-hand input arguments.
plhs	An array of left-hand output arguments.
nrhs	The number of right-hand arguments, or the size of the `prhs` array.
nlhs	The number of left-hand arguments, or the size of the `plhs` array.

Both `prhs` and `plhs` are declared as type `mxArray *`, which means they point to MATLAB arrays. They are vectors that contain pointers to the arguments of the MEX-file.

You can think of the name prhs as representing the "parameters, right-hand side," that is, the input parameters. Likewise, plhs represents the "parameters, left-hand side," or output parameters.

Creating and Using Source Files

It is good practice to write the gateway routine to call a "Computational Routine" on page 4-5; however, this is not required. The computational code can be part of the gateway routine. If you use both gateway and computational routines, they can be combined in one source file or in separate files. If you use separate files, the gateway routine must be the first source file listed in the mex command.

The name of the file containing your gateway routine is important, as explained in "Naming the MEX-File" on page 5-5.

Name your Fortran source file with an uppercase .F file extension.

The Difference Between .f and .F Files. Fortran compilers assume source files using a lowercase .f file extension have been preprocessed. On most platforms, mex makes sure the file is preprocessed regardless of the file extension. However, on Apple Macintosh platforms, mex cannot force preprocessing. Use an uppercase .F file extension to ensure your Fortran MEX-file is platform independent.

Using MATLAB Libraries

The MATLAB C and Fortran API Reference describes functions you can use in your gateway and computational routines that interact with MATLAB programs and the data in the MATLAB workspace. The mx prefixed functions provide access methods for manipulating MATLAB arrays. The mex prefixed functions perform operations in the MATLAB environment.

Required Header Files

To use the functions in the C and Fortran Reference library you must include the mex header, which declares the entry point and interface routines. Put this statement in your source file:

```
#include "mex.h"
```

In addition, Fortran MEX-files require the `fintrf.h` header file, which is used by the `mwPointer` preprocessor macro. Put this statement in your Fortran source file:

```
#include "fintrf.h"
```

Naming the MEX-File

The binary MEX-file name, and hence the name of the function you use in MATLAB, is the name of the source file containing your gateway routine.

The file extension of the binary MEX-file is platform-dependent. You find the file extension using the `mexext` function, which returns the value for the current machine.

Computational Routine

The *computational routine* contains the code for performing the computations you want implemented in the binary MEX-file. Computations can be numerical computations as well as inputting and outputting data. The gateway calls the computational routine as a subroutine.

The programming requirements described in "Creating and Using Source Files" on page 4-4, "Using MATLAB Libraries" on page 4-4, and "Required Header Files" on page 4-4 may also apply to your computational routine.

Preprocessor Macros

The MATLAB *preprocessor macros* `mwSize` and `mwIndex` are used in the `mx` and `mex` functions for cross-platform flexibility. `mwSize` represents size values, such as array dimensions and number of elements. `mwIndex` represents index values, such as indices into arrays.

MATLAB has an additional preprocessor macro for Fortran files, `mwPointer`. MATLAB uses a unique data type, the `mxArray`. Because there is no way to create a new data type in Fortran, MATLAB passes a special identifier, created by the `mwPointer` preprocessor macro, to a Fortran program. This is how you get information about an `mxArray` in a native Fortran data type. For example, you can find out the size of the `mxArray`, determine whether or not

it is a string, and look at the contents of the array. Use `mwPointer` to build platform-independent code.

The Fortran preprocessor converts `mwPointer` to `integer*4` when building binary MEX-files on 32-bit platforms and to `integer*8` when building on 64-bit platforms.

Note Declaring a pointer to be the incorrect size may cause your program to crash.

Using the Fortran %val Construct

The Fortran `%val(`*arg*`)` construct specifies that an argument, *arg*, is to be passed by value, instead of by reference. The `%val` construct is supported by most, but not all, Fortran compilers.

If your compiler does not support the `%val` construct, you must copy the array values into a temporary true Fortran array using the `mxCopy*` routines (e.g., `mxCopyPtrToReal8`).

A %val Construct Example

If your compiler supports the `%val` construct, you can use routines that point directly to the data (i.e., the pointer returned by `mxGetPr` or `mxGetPi`). You can use `%val` to pass this pointer's contents to a subroutine, where it is declared as a Fortran double-precision matrix.

For example, consider a gateway routine that calls its computational routine, `yprime`, by:

```
call yprime(%val(yp), %val(t), %val(y))
```

If your Fortran compiler does not support the `%val` construct, you would replace the call to the computational subroutine with:

```
C Copy array pointers to local arrays.
      call mxCopyPtrToReal8(t, tr, 1)
      call mxCopyPtrToReal8(y, yr, 4)
C
```

```
C Call the computational subroutine.
      call yprime(ypr, tr, yr)
C
C Copy local array to output array pointer.
      call mxCopyReal8ToPtr(ypr, yp, 4)
```

You must also add the following declaration line to the top of the gateway routine:

```
real*8 ypr(4), tr, yr(4)
```

Note that if you use `mxCopyPtrToReal8` or any of the other `mxCopy*` routines, the size of the arrays declared in the Fortran gateway routine must be greater than or equal to the size of the inputs to the MEX-file coming in from MATLAB. Otherwise, `mxCopyPtrToReal8` does not work correctly.

Data Flow in MEX-Files

The following examples illustrate data flow in MEX-files:

Showing Data Input and Output

Suppose your MEX-file `myFunction` has 2 input arguments and 1 output argument. The MATLAB syntax is `[X] = myFunction(Y, Z)`. To call `myFunction` from MATLAB, type:

```
X = myFunction(Y, Z);
```

The MATLAB interpreter calls `mexFunction`, the gateway routine to `myFunction`, with the following arguments:

```
nlhs = 1

nrhs = 2
```


Your input is `prhs`, a 2-element C array (`nrhs = 2`). The first element is a pointer to an `mxArray` named `Y` and the second element is a pointer to an `mxArray` named `Z`.

Your output is `plhs`, a 1-element C array (`nlhs = 1`) where the single element is a `null` pointer. The parameter `plhs` points at nothing because the output `X` is not created until the subroutine executes.

The gateway routine creates the output array and sets a pointer to it in `plhs[0]`. If `plhs[0]` is left unassigned and you assign an output value to the function when you call it, MATLAB generates an error stating that no output was assigned.

Note It is possible to return an output value even if `nlhs = 0`. This corresponds to returning the result in the `ans` variable.

Gateway Routine Data Flow Diagram

The following MEX Cycle diagram shows how inputs enter a MEX-file, what functions the gateway routine performs, and how outputs return to MATLAB.

In this example, the syntax of the MEX-file `func` is `[C, D] = func(A,B)`. In the figure, a call to `func` tells MATLAB to pass variables `A` and `B` to your MEX-file. `C` and `D` are left unassigned.

The gateway routine `func.F` uses the `mxCreate*` functions to create the MATLAB arrays for your output arguments. It sets `plhs[0]` and `plhs[1]`

to the pointers to the newly created MATLAB arrays. It uses the `mxGet*` functions to extract your data from your input arguments `prhs[0]` and `prhs[0]`. Finally, it calls your computational routine, passing the input and output data pointers as function parameters.

On return to MATLAB, `plhs[0]` is assigned to `C` and `plhs[1]` is assigned to `D`.

Inputs

```
integer B
B = prhs(2)
```

```
integer A
A = prhs(1)
```

MATLAB

A call to MEX-file `func`:

`[C,D]=func(A,B)`

tells MATLAB to pass variables A and B to your MEX-file. C and D are left unassigned.

func.F

```
subroutine mexFunction(
  nlhs, plhs, nrhs, prhs)
integer plhs(*), prhs(*),nlhs, nrhs
```

In the gateway routine:

- Use the `mxCreate` functions to create the MATLAB arrays for your output arguments. Set `plhs(1),(2),...` to the pointers to the newly created MATLAB arrays.

- Use the `mxGet` functions to extract your data from `prhs(1),(2),...`.

- Call your Fortran subroutine passing the input and output data pointers as function parameters using `%val`.

MATLAB

On return from MEX-file `func`:

`[C,D]=func(A,B)`

`plhs(1)` is assigned to C and `plhs(2)` is assigned to D.

```
integer D
D = plhs(2)
```

```
integer C
C = plhs(1)
```

Outputs
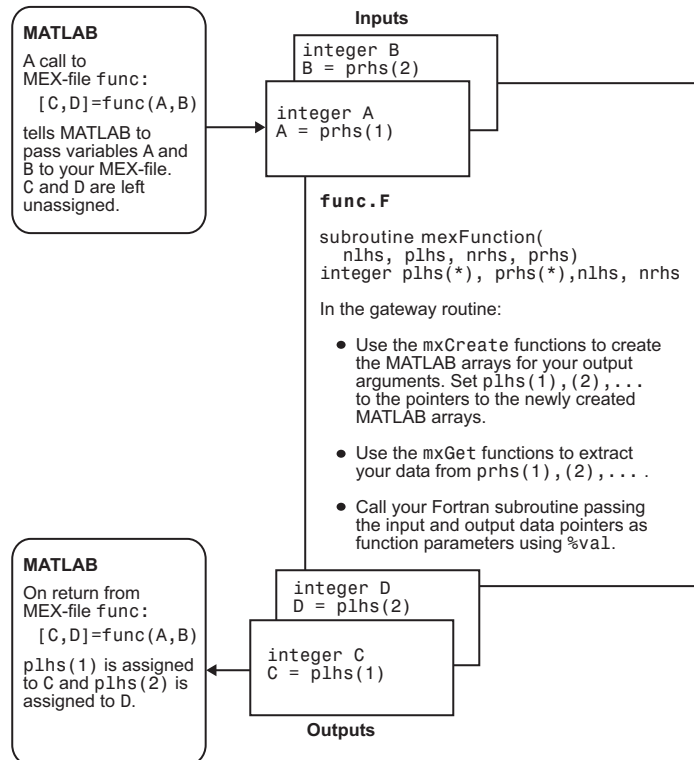
Fortran MEX Cycle

MATLAB Example yprime.F

Let's look at an example, `yprime.F`, found in your *matlabroot*/`extern/examples/mex/` directory. ("Building MEX-Files" on page 3-22 explains how to create the binary MEX-file.) It's calling syntax is `[YP] = YPRIME(T,Y)`, where `T` is an integer and `Y` is a vector with 4 elements. For `T=1` and `Y=1:4`, when you type:

```
      yprime(T,Y)
```

MATLAB displays:

```
ans =
     2.0000    8.9685    4.0000    -1.0947
```

The gateway routine should validate the input arguments. This step includes checking the number, type, and size of the input arrays as well as examining the number of output arrays. If the inputs are not valid, call mexErrMsgTxt. For example:

```
C
C CHECK FOR PROPER NUMBER OF ARGUMENTS
C
      IF (NRHS .NE. 2) THEN
        CALL MEXERRMSGTXT('YPRIME requires two input arguments')
      ELSEIF (NLHS .GT. 1) THEN
        CALL MEXERRMSGTXT('YPRIME requires one output argument')
      ENDIF
C
C CHECK THE DIMENSIONS OF Y.  IT CAN BE 4 X 1 OR 1 X 4.
C
      M = MXGETM(PRHS(2))
      N = MXGETN(PRHS(2))
C
      IF ((MAX(M,N) .NE. 4) .OR. (MIN(M,N) .NE. 1)) THEN
        CALL MEXERRMSGTXT('YPRIME requires that Y be a 4 x 1 vector')
      ENDIF
```

To create MATLAB arrays, call any of the mxCreate* functions, like mxCreateDoubleMatrix, mxCreateSparse, or mxCreateString. If it needs them, the gateway routine can call mxCalloc to allocate temporary work arrays for the computational routine. In this example:

```
C
C CREATE A MATRIX FOR RETURN ARGUMENT
C
      PLHS(1) = MXCREATEDOUBLEMATRIX(M,N,0)
```

In the gateway routine, you access the data in mxArray and manipulate it in your computational subroutine. For example, the expression mxGetPr(prhs[0]) returns a pointer of type double * to the real data in the mxArray pointed to by prhs[0].You can then use this pointer like any other pointer of type double * in Fortran. For example:

```
C
C ASSIGN POINTERS TO THE VARIOUS PARAMETERS
C
      YPP = MXGETPR(PLHS(1))
C
      TP = MXGETPR(PRHS(1))
      YP = MXGETPR(PRHS(2))
C
C COPY RIGHT HAND ARGUMENTS TO LOCAL ARRAYS OR VARIABLES
      NEL = 1
      CALL MXCOPYPTRTOREAL8(TP, RTP, NEL)
      NEL = 4
      CALL MXCOPYPTRTOREAL8(YP, RYP, NEL)
```

In this example, a computational routine, yprime, performs the calculations:

```
C
C DO THE ACTUAL COMPUTATIONS IN A SUBROUTINE
C       CREATED ARRAYS.
C
      CALL YPRIME(RYPP,RTP,RYP)
```

After calling your computational routine from the gateway, you can set a pointer of type mxArray to the data it returns. MATLAB recognizes the output from your computational routine as the output from the binary MEX-file.

```
C
C COPY OUTPUT WHICH IS STORED IN LOCAL ARRAY TO MATRIX OUTPUT
      NEL = 4
      CALL MXCOPYREAL8TOPTR(RYPP, YPP, NEL)
```

When a binary MEX-file completes its task, it returns control to MATLAB. Any MATLAB arrays that are created by the MEX-file but are not returned to MATLAB through the left-hand side arguments are automatically destroyed.

In general, we recommend that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism.

Examples of Fortran Source MEX-Files

Introduction

The MATLAB API provides a set of Fortran routines that handle double-precision data and strings in MATLAB. For each data type, there is a specific set of functions that you can use for data manipulation.

Source code for the examples in this chapter are located in the *matlabroot*/extern/examples/refbook directory of your MATLAB installation. To build these examples, make sure you have a Fortran compiler selected using the mex -setup command. Then at the MATLAB command prompt, type:

```
mex filename.F
```

where *filename* is the name of the example.

This section looks at source code for the examples. Unless otherwise specified, the term "MEX-file" refers to a source file.

A First Example — Passing a Scalar

Let's look at a simple example of Fortran code and its MEX-file equivalent. Here is a Fortran computational routine that takes a scalar and doubles it:

```
      subroutine timestwo(y, x)
      real*8 x, y
C
      y = 2.0 * x
      return
      end
```

To see the same function written in the MEX-file format (`timestwo.F`), open the file in the MATLAB Editor.

To build this example, at the command prompt type:

```
  mex timestwo.F
```

This command creates the binary MEX-file called `timestwo` with an extension corresponding to the machine type on which you're running. You can now call `timestwo` as if it were an M-function. Type:

```
  x = 2;
  y = timestwo(x)
```

MATLAB displays:

```
  y =
      4
```

Passing Strings

Passing strings from MATLAB to a Fortran MEX-file is straightforward. The program `revord.F` accepts a string and returns the characters in reverse order. To see the example `revord.F`, open the file in the MATLAB Editor.

After checking for the correct number of inputs, the gateway routine `mexFunction` verifies that the input was a row vector string. It then finds the size of the string and places the string into a Fortran character array. Note that in the case of character strings, it is not necessary to copy the data into a Fortran character array using `mxCopyPtrToCharacter`. In fact,

`mxCopyPtrToCharacter` works only with MAT-files. For more information, see "Using MAT-Files" on page 1-2.

To build this example, at the command prompt type:

```
mex revord.F
```

Type:

```
x = 'hello world';
y = revord(x)
```

MATLAB displays:

```
y =

dlrow olleh
```

Passing Arrays of Strings

Passing arrays of strings adds a complication to the example "Passing Strings" on page 5-14. Because MATLAB stores elements of a matrix by column instead of by row, the size of the string array must be correctly defined in the Fortran MEX-file. The key point is that the row and column sizes as defined in MATLAB must be reversed in the Fortran MEX-file. Consequently, when returning to MATLAB, the output matrix must be transposed.

This example places a string array/character matrix into MATLAB as output arguments rather than placing it directly into the workspace.

To build this example, at the command prompt type:

```
mex passstr.F
```

Type:

```
passstr;
```

to create the 5-by-15 `mystring` matrix. You need to do some further manipulation. The original string matrix is 5-by-15. Because of the way MATLAB reads and orients elements in matrices, the size of the matrix must be defined as M=15 and N=5 in the MEX-file. After the matrix is put into

MATLAB, the matrix must be transposed. The program `passstr.F` illustrates how to pass a character matrix. To see the code `passstr.F`, open the file in the MATLAB Editor.

Type:

```
passstr
```

MATLAB displays:

```
ans =

MATLAB
The Scientific
Computing
Environment
    by TMW, Inc.
```

Passing Matrices

In MATLAB, you can pass matrices into and out of MEX-files written in Fortran. You can manipulate the MATLAB arrays by using `mxGetPr` and `mxGetPi` to assign pointers to the real and imaginary parts of the data stored in the MATLAB arrays. You can create new MATLAB arrays from within your MEX-file by using `mxCreateDoubleMatrix`.

The example `matsq.F` takes a real 2-by-3 matrix and squares each element. To see the source code, open the file in MATLAB Editor.

After performing error checking to ensure that the correct number of inputs and outputs was assigned to the gateway subroutine and to verify the input was in fact a numeric matrix, `matsq.F` creates a matrix. The matrix is copied to a Fortran matrix using `mxCopyPtrToReal8`. Now the computational subroutine can be called, and the return argument is placed into `y_pr`, the pointer to the output, using `mxCopyReal8ToPtr`.

To build this example, at the command prompt type:

```
mex matsq.F
```

For a 2-by-3 real matrix, type:

```
x = [1 2 3; 4 5 6];
y = matsq(x)
```

MATLAB displays:

```
y =
    1      4      9
   16     25     36
```

Passing Two or More Inputs or Outputs

The `plhs` and `prhs` parameters (see "The Components of a Fortran MEX-File" on page 5-2) are vectors containing pointers to the left-hand side (output) variables and right-hand side (input) variables. `plhs(1)` contains a pointer to the first left-hand side argument, `plhs(2)` contains a pointer to the second left-hand side argument, and so on. Likewise, `prhs(1)` contains a pointer to the first right-hand side argument, `prhs(2)` points to the second, and so on.

The example `xtimesy.F` multiplies an input scalar times an input scalar or matrix. To see the source code, open the file in MATLAB Editor.

As this example shows, creating MEX-file gateways that handle multiple inputs and outputs is straightforward. All you need to do is keep track of which indices of the vectors `prhs` and `plhs` correspond to which input and output arguments of your function. In this example, the input variable `x` corresponds to `prhs(1)` and the input variable `y` to `prhs(2)`.

To build this example, at the command prompt type:

```
mex xtimesy.F
```

For an input scalar x and a real 3-by-3 matrix, type:

```
x = 3; y = ones(3);
z = xtimesy(x, y)
```

MATLAB displays:

```
z =
     3     3     3
     3     3     3
     3     3     3
```

Handling Complex Data

MATLAB stores complex double-precision data as two vectors of numbers—one vector contains the real data and the other contains the imaginary data. The functions mxCopyPtrToComplex16 and mxCopyComplex16ToPtr copy MATLAB data to a native complex*16 Fortran array.

The example convec.F takes two complex vectors (of length 3) and convolves them. To see the source code, open the file in the MATLAB Editor.

To build this example, at the command prompt type:

```
mex convec.F
```

Enter the following at the command prompt:

```
x = [3 - 1i, 4 + 2i, 7 - 3i];
y = [8 - 6i, 12 + 16i, 40 - 42i];
```

Type:

```
z = convec(x, y)
```

MATLAB displays:

```
z =

   1.0e+02 *

  Columns 1 through 4
```

```
    0.1800 - 0.2600i   0.9600 + 0.2800i   1.3200 - 1.4400i
    3.7600 - 0.1200i

  Column 5

    1.5400 - 4.1400i
```

which agrees with the results the built-in MATLAB function `conv.m` produces.

Dynamically Allocating Memory

To allocate memory dynamically in a Fortran MEX-file, use `%val`. (See "Using the Fortran %val Construct" on page 5-6.) The example `dblmat.F` takes an input matrix of real data and doubles each of its elements. To see the source code, open the file in the MATLAB Editor. `compute.F` is the subroutine `dblmat` calls to double the input matrix. (Open the file in the MATLAB Editor.)

To build this example, at the command prompt type:

```
  mex dblmat.F compute.F
```

For the 2-by-3 matrix, type:

```
  x = [1 2 3; 4 5 6];
  y = dblmat(x)
```

MATLAB displays:

```
  y =
      2     4     6
      8    10    12
```

Note The `dblmat.F` example, as well as `fulltosparse.F` and `sincall.F`, are split into two parts, the gateway and the computational subroutine, because of restrictions in some compilers.

Handling Sparse Matrices

MATLAB provides a set of functions that allow you to create and manipulate sparse matrices. There are special parameters associated with sparse matrices, namely `ir`, `jc`, and `nzmax`. For information on how to use these parameters and how MATLAB stores sparse matrices in general, see "Sparse Matrices" on page 3-20.

Note Sparse array indexing is zero-based, not one-based.

The `fulltosparse.F` example illustrates how to populate a sparse matrix. To see the source code, open the file in the MATLAB Editor. `loadsparse.F` is the subroutine `fulltosparse` calls to fill the `mxArray` with the sparse data. (Open the file in the MATLAB Editor.)

To build this example, at the command prompt type:

```
mex fulltosparse.F loadsparse.F
```

At the command prompt, typing:

```
full = eye(5)
full =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

creates a full, 5-by-5 identity matrix. Using `fulltosparse` on the full matrix produces the corresponding sparse matrix:

```
spar = fulltosparse(full)
spar =
   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
```

Calling Functions from Fortran MEX-Files

You can call MATLAB functions, operators, M-files, and even other binary MEX-files from within your Fortran source code by using the API function `mexCallMATLAB`. The `sincall.F` example creates an `mxArray`, passes various pointers to a subfunction to acquire data, and calls `mexCallMATLAB` to calculate the sine function and plot the results. To see the source code, open the file in the MATLAB Editor. `fill.F` is the subroutine `sincall` calls to fill the `mxArray` with data. (Open the file in the MATLAB Editor.)

It is possible to use `mexCallMATLAB` (or any other API routine) from within your computational Fortran subroutine. Note that you can only call most MATLAB functions with double-precision data. M-functions that perform computations, such as `eig`, do not work correctly with data that is not double precision.

To build this example, at the command prompt type:

```
mex sincall.F fill.F
```

Running this example:

```
sincall
```

displays the results:
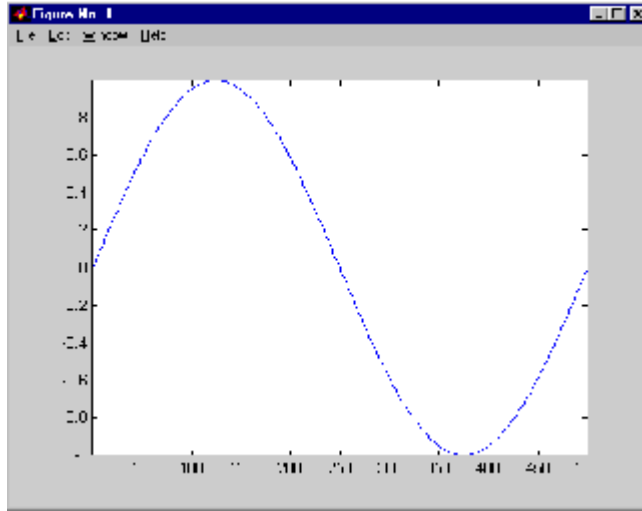
Note You can generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. See the following example.

This example creates an M-file that returns two variables but only assigns one of them a value:

```
function [a,b]=foo[c]
a=2*c;
```

MATLAB displays the following warning:

```
Warning: One or more output arguments not assigned during call to
'foo'.
```

If you then call `foo` using `mexCallMATLAB`, the unassigned output variable is now of type `mxUNKNOWN_CLASS`.

Advanced Topics

In this section...

Help Files

Because the MATLAB interpreter chooses the binary MEX-file when both an M-file and a MEX-file with the same name are encountered in the same directory, it is possible to use M-files for documenting the behavior of your binary MEX-files. The `help` command automatically finds and displays the appropriate M-file when help is requested and the interpreter finds and executes the corresponding binary MEX-file when the function is invoked.

Linking Multiple Files

You can combine multiple source files, object files, and file libraries to build a binary MEX-file. To do this, list the additional files, with their file extensions, separated by spaces. The name of the resulting MEX-file is the name of the first file in the list.

The following command combines multiple files of different types into a binary MEX-file called `circle.ext`, where *ext* is the extension corresponding to the current platform:

```
mex circle.F square.o rectangle.F shapes.o
```

You may find it useful to use a software development tool like `MAKE` to manage MEX-file projects involving multiple source files. Simply create a `MAKEFILE` that contains a rule for producing object files from each of your source files, and then invoke the `mex` build script to combine your object files into a binary MEX-file. This way you can ensure that your source files are recompiled only when necessary.

> **Note** On Linux[14] systems, you must use the `-fortran` switch to the `mex` script if you are linking Fortran objects.

Workspace for MEX-File Functions

Unlike M-file functions, MEX-file functions (binary MEX-files) do not have their own variable workspace. MEX-file functions operate in the caller's workspace. `mexEvalString` evaluates the string in the caller's workspace. In addition, you can use the `mexGetVariable` and `mexPutVariable` routines to get and put variables into the caller's workspace.

Handling Large mxArrays

Binary MEX-files built on 64-bit platforms can handle 64-bit `mxArrays`. These large data arrays can have up to $2^{48}-1$ elements. The maximum number of elements a sparse `mxArray` can have is $2^{48}-2$.

Using the following instructions creates platform-independent binary MEX-files as well.

Your system configuration can impact the performance of MATLAB. The 64-bit processor requirement enables you to create the `mxArray` and access data in it. However, your system's memory, in particular the size of RAM and virtual memory, determine the speed at which MATLAB processes the `mxArray`. The more memory available, the faster the processing.

The amount of RAM also limits the amount of data you can process at one time in MATLAB. For guidance on memory issues, see "Strategies for Efficient Use of Memory" in the Programming Fundamentals documentation. Memory management within source MEX-files can have special considerations, as described in "Memory Management" on page 4-30.

Using the 64-Bit API

To work with a 64-bit `mxArray`, your source code must comply with the 64-bit API, which consists of the functions in the following table.

---

14. Linux is a registered trademark of Linus Torvalds.

mxCalcSingleSubscript	mxCreateSparseLogicalMatrix
mxCalloc	mxCreateStructArray
mxCopyCharacterToPtr	mxCreateStructMatrix
mxCopyComplex16ToPtr	mxGetCell
mxCopyComplex8ToPtr	mxGetDimensions
mxCopyInteger1ToPtr	mxGetElementSize
mxCopyInteger2ToPtr	mxGetField
mxCopyInteger4ToPtr	mxGetFieldByNumber
mxCopyPtrToCharacter	mxGetIr
mxCopyPtrToComplex16	mxGetJc
mxCopyPtrToComplex8	mxGetM
mxCopyPtrToInteger1	mxGetN
mxCopyPtrToInteger2	mxGetNumberOfDimensions
mxCopyPtrToInteger4	mxGetNumberOfElements
mxCopyPtrToPtrArray	mxGetNzmax
mxCopyPtrToReal4	mxGetProperty
mxCopyPtrToReal8	mxGetString
mxCopyReal4ToPtr	mxMalloc
mxCopyReal8ToPtr	mxRealloc
mxCreateCellArray	mxSetCell
mxCreateCellMatrix	mxSetDimensions
mxCreateCharArray	mxSetField
mxCreateCharMatrixFromStrings	mxSetFieldByNumber
mxCreateDoubleMatrix	mxSetIr
mxCreateLogicalArray	mxSetJc
mxCreateLogicalMatrix	mxSetM
mxCreateNumericArray	mxSetN

```
mxCreateNumericMatrix          mxSetNzmax

mxCreateSparse                 mxSetProperty
```

Functions in this API use the `mwIndex`, `mwSize`, and `mwPointer` preprocessor macros. For information about using these macros, see "Required Header Files" on page 5-4.

Building the Binary MEX-File
Use the `mex` build script option `-largeArrayDims` with the 64-bit API.

Caution Using Negative Values
When using the 64-bit API, `mwSize` and `mwIndex` are equivalent to `size_t` in C or `INTEGER*8` in Fortran. These types are unsigned, unlike `int` and `INTEGER*4`, which are the types used in the 32-bit API. Be careful not to pass any negative values to functions that take `mwSize` or `mwIndex` arguments. Do not cast negative `int` or `INTEGER*4` values to `mwSize` or `mwIndex`; the returned value can not be predicted. Instead, change your code to avoid using negative values.

Building Cross-Platform Applications
If you develop cross-platform applications (programs that can run on both 32- and 64-bit architectures), you must pay attention to the upper limit of values you use for `mwSize` and `mwIndex`. The 32-bit application reads these values and assigns them to variables declared as `int` in C or `INTEGER*4` in Fortran. Be careful to avoid assigning a large `mwSize` or `mwIndex` value to an `int`, `INTEGER*4`, or other variable that might be too small.

Memory Management
When a binary MEX-file returns control to MATLAB, it returns the results of its computations in the output arguments—the `mxArrays` contained in the left-hand side arguments `plhs[]`. MATLAB destroys any `mxArray` created by the MEX-file that is not in this argument list. In addition, MATLAB frees any memory that was allocated in the MEX-file using the `mxCalloc`, `mxMalloc`, or `mxRealloc` functions.

Consequently, any misconstructed arrays left over at the end of a binary MEX-file's execution have the potential to cause memory errors.

In general, we recommend that MEX-file functions destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient to perform this cleanup in the source MEX-file than to rely on the automatic mechanism.For additional information on memory management techniques, see the sections "Memory Management" on page 4-30 in Creating C Language MEX-Files and "Memory Management Issues" on page 3-55.

Debugging Fortran Source MEX-Files

In this section...
"Notes on Debugging" on page 5-28
"Debugging on Microsoft Windows Platforms" on page 5-28
"Debugging on Linux Platforms" on page 5-28

Notes on Debugging

The examples show how to debug timestwo.F, found in your *matlabroot*/extern/examples/refbook/ directory.

Binary MEX-files built with the -g option do not execute on other computers because they rely on files that are not distributed with MATLAB software. Refer to the "Calling C and Fortran Programs from MATLAB" topic "Troubleshooting MEX-Files" on page 3-47 for additional information on isolating problems with MEX-files.

Debugging on Microsoft Windows Platforms

For MEX-files compiled with any version of the Intel Visual Fortran compiler, you can use the debugging tools found in your version of Microsoft Visual Studio. Refer to the "Creating C Language MEX-Files" topic "Debugging on the Microsoft Windows Platforms" on page 4-48 for instructions on using this debugger.

For information on debugging MEX-files compiled with other MATLAB supported compilers, see Technical Note 1605, MEX-files Guide, at http://www.mathworks.com/support/tech-notes/1600/1605.html.

Debugging on Linux Platforms

The MATLAB supported Fortran compiler g95 has a -g option for building binary MEX-files with debug information. Such files can be used with gdb, the GNU Debugger. This section describes using gdb.

For information on debugging MEX-files compiled with other MATLAB supported compilers, see Technical Note 1605, MEX-files Guide, at `http://www.mathworks.com/support/tech-notes/1600/1605.html`.

GNU Debugger gdb

In this example, the MATLAB command prompt >> is shown in front of MATLAB commands, and `linux>` represents a Linux[15] prompt; your system may show a different prompt. The debugger prompt is `<gdb>`.

1 To compile the source MEX-file, type:

```
linux> mex -g timestwo.F
```

On a Linux 32–bit platform, this command creates the executable file `timestwo.mexglx`.

2 At the Linux prompt, start the gdb debugger using the `matlab -D` option:

```
linux> matlab -Dgdb
```

3 Start MATLAB without the Java Virtual Machine (JVM) by using the `-nojvm` startup flag:

```
<gdb> run -nojvm
```

4 In MATLAB, enable debugging with the `dbmex` function and run your binary MEX-file:

```
>> dbmex on
>> y = timestwo(4)
```

5 At this point, you are ready to start debugging.

It is often convenient to set a breakpoint at `mexFunction` so you stop at the beginning of the gateway routine.

---

15. Linux is a registered trademark of Linus Torvalds.

> **Note** The function name may be slightly altered by the compiler (e.g., it may have an underscore appended). To determine how this symbol appears in a given MEX-file, use the Linux command nm. For example:
>
> ```
> linux> nm timestwo.mexglx | grep -i mexfunction
> ```
>
> The operating system responds with something like:
>
> ```
> 0000091c T mexfunction_
> ```
>
> Use mexfunction_ in the breakpoint statement. Be sure to use the correct case.

```
<gdb> break mexfunction_
<gdb> continue
```

6 Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.

To proceed from a breakpoint, type continue:

```
<gdb> continue
```

7 After stopping at the last breakpoint, type:

```
<gdb> continue
```

timestwo finishes and MATLAB displays:

```
y =

     8
```

8 From the MATLAB prompt you can return control to the debugger by typing:

```
>> dbmex stop
```

Or, if you are finished running MATLAB, type:

```
>> quit
```

9 When you are finished with the debugger, type:

```
<gdb> quit
```

You return to the Linux prompt.

Refer to the documentation provided with your debugger for more information on its use.

6

Calling MATLAB Software from C and Fortran Programs

The MATLAB engine library contains routines that allow you to call MATLAB software from your own programs, thereby employing MATLAB as a computation engine. Engine programs are C or Fortran programs that communicate with a separate MATLAB process via pipes, on UNIX[16] systems, and through a Microsoft Component Object Model (COM) interface, on Microsoft Windows systems. MATLAB provides a library of functions that allows you to start and end the MATLAB process, send data to and from MATLAB, and send commands to be processed in MATLAB.

---

16. UNIX is a registered trademark of The Open Group in the United States and other countries.

Using the MATLAB Engine to Call MATLAB Software from C and Fortran Programs

In this section...
"Introduction" on page 6-2
"The Engine Library" on page 6-3
"GUI-Intensive Applications" on page 6-4

Introduction

Some of the things you can do with the MATLAB engine are

- Call a math routine, for example, to invert an array or to compute an FFT from your own program. When employed in this manner, MATLAB is a powerful and programmable mathematical subroutine library.

- Build an entire system for a specific task, for example, radar signature analysis or gas chromatography, where the front end (GUI) is programmed in C and the back end (analysis) is programmed in MATLAB, thereby shortening development time.

The MATLAB engine operates by running in the background as a separate process from your own program. This offers several advantages:

- On UNIX[17] systems, the engine can run on your machine, or on any other UNIX machine on your network, including machines of a different architecture. This allows you to implement a user interface on your workstation and perform the computations on a faster machine located elsewhere on your network. The description of the engOpen function offers further information.

- Instead of requiring your program to link to the entire MATLAB program (a substantial amount of code), it links to a smaller engine library.

---

17. UNIX is a registered trademark of The Open Group in the United States and other countries.

Note To run MATLAB engine on the UNIX platform, you must have the C shell `csh` installed at `/bin/csh`.

The Engine Library

The engine library contains the following routines for controlling the computation engine. The names begin with the three-letter prefix `eng`.

C Engine Routines

Function	Purpose
engOpen	Start up MATLAB engine
engClose	Shut down MATLAB engine
engGetVariable	Get a MATLAB array from the engine
engPutVariable	Send a MATLAB array to the engine
engEvalString	Execute a MATLAB command
engOutputBuffer	Create a buffer to store MATLAB text output
engOpenSingleUse	Start a MATLAB engine session for single, nonshared use
engGetVisible	Determine visibility of MATLAB engine session
engSetVisible	Show or hide MATLAB engine session

Fortran Engine Routines

Function	Purpose
engOpen	Start up MATLAB engine
engClose	Shut down MATLAB engine
engGetVariable	Get a MATLAB array from the engine
engPutVariable	Send a MATLAB array to the engine
engEvalString	Execute a MATLAB command
engOutputBuffer	Create a buffer to store MATLAB text output

The engine also uses the `mx`–prefixed API routines discussed in Chapter 4, "Creating C Language MEX-Files" and Chapter 5, "Creating Fortran MEX-Files".

Communicating with MATLAB Software

On UNIX systems, the engine library communicates with the engine using pipes, and, if needed, `rsh` for remote execution. On Microsoft Windows systems, the engine library communicates with the engine using a Component Object Model (COM) interface. For more information, see Chapter 8, "COM Support for MATLAB Software".

GUI-Intensive Applications

If you have graphical user interface (GUI) intensive applications that execute a lot of callbacks through the MATLAB engine, you should force these callbacks to be evaluated in the context of the base workspace. Use `evalin` to specify that the base workspace be used in evaluating the callback expression, as follows:

```
engEvalString(ep, "evalin('base', expression)")
```

Specifying the base workspace in this manner ensures MATLAB processes the callback correctly and returns results for that call.

This does not apply to computational applications that do not execute callbacks.

Examples of Calling Engine Functions

Overview

This section describes steps you must follow when using the engine functions. For example, before using `engPutVariable`, you must create a matrix and populate it.

After reviewing these examples, follow the instructions in "Compiling and Linking MATLAB Engine Programs" on page 6-10 to build the application and test it. You can test that your system is properly configured for engine applications by building and running an application.

Calling MATLAB Software from a C Application

The program, `engdemo.c`, illustrates how to call the engine functions from a stand alone C program. For the Microsoft Windows version of this program, see `engwindemo.c` in the *matlabroot*`\extern\examples\eng_mat` directory. *matlabroot* is the MATLAB root directory. Engine examples, like the MAT-file examples, are located in the `eng_mat` directory *matlabroot*`\extern\examples\eng_mat`.

To see `engdemo.c`, open the file in the MATLAB Editor.

To see the Windows version `engwindemo.c`, open the file.

The first part of this program launches MATLAB and sends it data. MATLAB analyzes the data and plots the results.
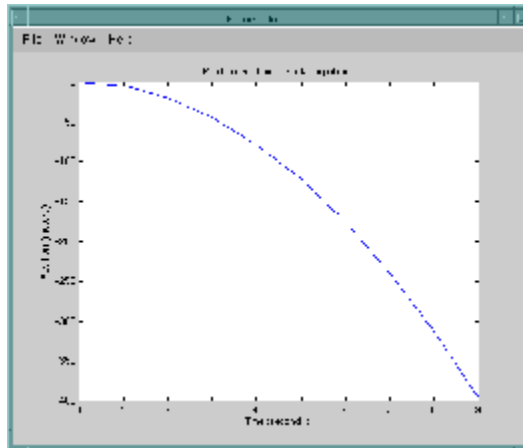
The program continues with:

```
Press Return to continue
```

Pressing **Return** continues the program:

```
Done for Part I.
Enter a MATLAB command to evaluate.  This command should
create a variable X.  This program will then determine
what kind of variable you created.
For example: X = 1:5
```

Entering X = 17.5 continues the program execution.

```
X = 17.5

X =

   17.5000


Retrieving X...
X is class double
Done!
```

Finally, the program frees memory, closes the MATLAB engine, and exits.

Calling MATLAB Software from a C++ Application

There is a C++ version of engdemo.c in the
matlabroot\extern\examples\eng_mat directory. To see engdemo.cpp, open
the file in the MATLAB Editor.

Calling MATLAB Software from a Fortran Application

The program, fengdemo.F, illustrates how to call the engine functions from a
stand alone Fortran program. To see the code, open the file in the MATLAB
Editor.

Executing this program launches MATLAB, sends it data, and plots the
results.
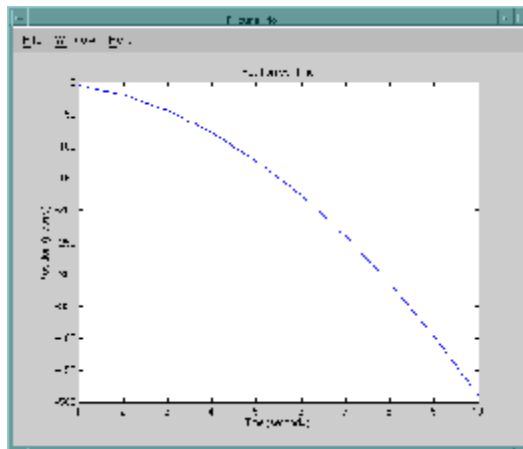
The program continues with:

```
Type 0 <return> to Exit
Type 1 <return> to continue
```

Entering 1 at the prompt continues the program execution:

```
1
 MATLAB computed the following distances:
   time(s)  distance(m)
   1.00     -4.90
```

```
2.00     -19.6
3.00     -44.1
4.00     -78.4
5.00     -123.
6.00     -176.
7.00     -240.
8.00     -314.
9.00     -397.
10.0     -490.
```

Finally, the program frees memory, closes the MATLAB engine, and exits.

Attaching to an Existing MATLAB Session

You can make an engine program attach to a MATLAB session that is already running by starting the MATLAB session with /Automation in the command line. When you make a call to engOpen, it connects to this existing session. You should only call engOpen once, because any engOpen calls now connect to this one MATLAB session.

The /Automation option also causes the command window to be minimized. You must open it manually.

Note For more information on the /Automation command-line argument, see "Additional Automation Server Information" on page 10-13. For information about the Component Object Model interfaces used by MATLAB, see "Introducing MATLAB COM Integration" on page 8-2.

For example,

1 Shut down any MATLAB sessions.

2 From the **Start** button on the Windows menu bar, click **Run**.

3 In the **Open** field, type:

```
d:\matlab\bin\win32\matlab.exe /Automation
```

or:

```
d:\matlab\bin\win64\matlab.exe /Automation
```

where `d:\matlab\bin\win32` or `d:\matlab\bin\win64` represents the path to the MATLAB executable.

4 Click **OK**. This starts MATLAB.

5 In MATLAB, change directories to *matlabroot*/extern/examples/eng_mat.

6 Compile the engwindemo.c example.

7 Run the engwindemo program by typing at the MATLAB prompt:

```
!engwindemo
```

This does not start another MATLAB session, but rather uses the MATLAB session that is already open.

Note On the UNIX[18] platform, you cannot make an engine program connect to an existingMATLAB session.

---

18. UNIX is a registered trademark of The Open Group in the United States and other countries.

Compiling and Linking MATLAB Engine Programs

Write Your Application

Write your application in C or Fortran using any of the engine routines to perform computations in MATLAB. For more information, see "Using the MATLAB Engine to Call MATLAB Software from C and Fortran Programs" on page 6-2 and "Examples of Calling Engine Functions" on page 6-5.

Check Required Libraries and Files

MATLAB requires the following files for building any engine application:

- "Third-Party Libraries" on page 6-11
- "Library Files Required by libeng" on page 6-11
- "Unicode Data Files" on page 6-12

Third-Party Libraries

Verify that the required libraries are installed. Use the following table to identify the path and library filename. Replace *libfile* with each of these filenames:

```
libeng
libmx
```

Operating System	Library Path and Filename
Linux[19]	*matlabroot*/bin/glnx86/*libfile*.so
64-bit Linux	*matlabroot*/bin/glnxa64/*libfile*.so
64-bit Sun Solaris SPARC	*matlabroot*/bin/sol64/*libfile*.so
AppleMacintosh (Intel)	*matlabroot*/bin/maci/*libfile*.dylib
Microsoft Windows	*matlabroot*\bin\win32*libfile*.dll
Windows x64	*matlabroot*\bin\win64*libfile*.dll

Library Files Required by libeng

The libeng library requires additional third-party library files. MATLAB uses these libraries to support Unicode character encoding and data compression in MAT-files.

These library files must reside in the same directory as libmx. You can determine what most of these libraries are using the platform-specific commands shown here.

Operating System	Library Path and Filename
All Linux, Solaris	ldd -d libeng.so
Macintosh	otool -L libeng.dylib
Windows	See the following instructions

---

19. Linux is a registered trademark of Linus Torvalds.

On Windows systems, the third-party product Dependency Walker can be used to diagnose errors related to loading and executing modules. Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module and builds a hierarchical tree diagram of all dependent modules. For each module found, it lists all the functions that are exported by that module, and which of those functions are actually being called by other modules. You can download the Dependency Walker utility from the following Web site:

```
http://www.dependencywalker.com/
```

See the Technical Support solution 1-2RQL4L for information on using the Dependency Walker:

```
http://www.mathworks.com/support/solutions/data/1-2RQL4L.html
```

Drag and drop the `libeng.dll` file into the Depends window.

Unicode Data Files

Verify that the appropriate Unicode data file is installed. Systems that order bytes in a big-endian manner use `icudt32b.dat`, and those that have little-endian ordering use `icudt32l.dat`.

Operating System	**Unicode File Path and Filename**
Linux	*matlabroot*/bin/glnx86/icudt32l.dat
64-bit Linux	*matlabroot*/bin/glnxa64/icudt32l.dat
64-bit Solaris SPARC	*matlabroot*/bin/sol64/icudt32b.dat
Macintosh (Intel)	*matlabroot*/bin/maci/icudt32b.dat
Windows	*matlabroot*\bin\win32\icudt32l.dat
Windows x64	*matlabroot*\bin\win64\icudt32l.dat

Note If you need to manipulate Unicode text directly in your application, the latest version of International Components for Unicode (ICU) is available online from the IBM Corporation Web site at `http://icu.sourceforge.net/download`.

Build the Application

Use the `mex` script to compile and link engine programs. `mex` has a set of switches you can use to modify the compile and link stages. The table MEX Script Switches on page 3-35 lists the available switches and their uses.

MEX Options File

MATLAB supplies an options file to facilitate building MEX applications. This file contains compiler-specific flags that correspond to the general compile, prelink, and link steps required on your system. If you want to customize the build process, you can modify this file.

Different options files are provided for UNIX and Windows operating systems.

Operating System	Default Options File
UNIX	*matlabroot*/bin/engopts.sh
Windows	*matlabroot*\bin\win32\mexopts*engmatopts.bat
Windows x64	*matlabroot*\bin\win64\mexopts*engmatopts.bat

On Windows systems, the options file depends on which compiler you use. The name of the options file is prefixed with a string representing the compiler and compiler version it is used with.

For example, to locate the options file on a Windows 32-bit system, type:

```
dir([matlabroot '\bin\win32\mexopts\*engmatopts.bat'])
```

If you need to modify the options file for your particular compiler, use the `mex` command with the `-v` switch to view the current compiler and linker settings, and then make the appropriate changes in the options file.

Build the Application

To build your engine application, use the `mex` script with the options filename and the name of your MEX-file.

UNIX Operating Systems. Enter the following command, where *mexfilename* is the name of your C or Fortran program. Enclose *mexfilename* in single quotation marks.

```
mex('-f', [matlabroot '/bin/engopts.sh'], mexfilename);
```

Alternatively, copy the options file to your current working directory, and then enter the following command:

```
mex -f engopts.sh mexfilename
```

Windows Operating Systems. Enter the following command, where *mexfilename* is the name of your C or Fortran program. Enclose *mexfilename* in single quotation marks. This example uses the Lcc compiler. Be sure to use the appropriate options file for your compiler.

```
mex('-f', [matlabroot ...
    '\bin\win32\mexopts\lccengmatopts.bat'], mexfilename);
```

Alternatively, copy the options file to your current working directory, and then enter the following command:

```
mex -f lccengmatopts.bat mexfilename
```

Set Run-Time Library Path

At run-time, you need to tell the system where the API shared libraries reside.

UNIX Operating Systems

Set the library path as follows for the C and Bourne shells. In the commands shown, replace the terms *envvar* and *pathspec* with the appropriate values from the table that follows.

To set the library path in the C shell, type:

```
setenv envvar pathspec
```

In the Bourne shell, type:

```
envvar = pathspec:envvar export envvar
```

Operating System	*envvar*	*pathspec*
Linux[20]	LD_LIBRARY_PATH	*matlabroot*/bin/glnx86: *matlabroot*/sys/os/glnx86
64-bit Linux	LD_LIBRARY_PATH	*matlabroot*/bin/glnxa64: *matlabroot*/sys/os/glnxa64
64-bit SunSolaris SPARC	LD_LIBRARY_PATH	*matlabroot*/bin/sol64: *matlabroot*/sys/os/sol64
Apple Macintosh (Intel)	DYLD_LIBRARY_PATH	*matlabroot*/bin/maci: *matlabroot*/sys/os/maci

Here is an example for a Solaris system for the C shell:

```
setenv LD_LIBRARY_PATH matlabroot/bin/sol64:matlabroot/sys/os/sol64
```

and for the Bourne shell:

```
LD_LIBRARY_PATH=matlabroot/bin/sol64:matlabroot/sys/os/sol64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

Place these commands in a startup script such as ~/.cshrc for the C shell or ~/.profile for the Bourne shell.

Windows Operating Systems

Set the Path environment variable to the path string returned by MATLAB in response to the following expression:

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

or:

```
[matlabroot '\bin\win64']
```

To set an environment variable in a Windows system, select **Start > Settings > Control Panel > System**. The System Properties dialog

---

20. Linux is a registered trademark of Linus Torvalds.

box is displayed. Click the **Advanced** tab, and then the **Environment Variables** button.

In the **System variables** panel scroll down until you find the Path variable. Click this variable to highlight it, and then click the **Edit** button to open the Edit System Variable dialog box. At the end of the path string, enter a semicolon and then the path string returned by evaluating the expression shown above in MATLAB. Click **OK** in the Edit System Variable dialog box, and in all remaining dialog boxes.

Select MATLAB Version

If you have multiple versions of MATLAB installed on your Windows operating system, the version you use to build your engine applications must be the first listed in your system PATH environment variable. If the MATLAB version you use to build the application is not the first listed on the path, you may see the following error:

```
Can't start MATLAB engine
```

For information about accessing the PATH environment variable through the Windows Control Panel, see the "Windows Operating Systems" on page 6-15 topic in Set Run-Time Library Path.

Register MATLAB Software as a COM Server

To run this program on a Windows operating system, you need to have MATLAB registered as a COM server on your system. This registration is part of the MATLAB installation and should have already been done for you as part of the installation. If, for some reason, the registration was not done or did not complete successfully, you may see the following error displayed when you try to run this example:

```
Can't start MATLAB engine
```

If you see this error, manually register MATLAB as a server by entering the following commands in a DOS command window:

```
cd matlabroot\bin\win32
matlab /regserver
```

or:

```
cd matlabroot\bin\win64
matlab /regserver
```

Close the MATLAB window that appears.

Test the Program

Test your application in MATLAB by typing:

```
!engwindemo
```

Example — Building an Engine Application on Windows System

MATLAB provides a demonstration program written in C that you can use to verify the build process on your computer. The demo file for Windows systems is engwindemo.c.

Copy the C language MEX-file engwindemo.c to your current working directory:

```
demofile = [matlabroot '\extern\examples\eng_mat\engwindemo.c'];
copyfile(demofile, '.');
```

Look in the \bin\win32\mexopts directory for the appropriate options file for the Lcc compiler. Use the following commands to build the executable file using this compiler:

```
optsfile = [matlabroot '\bin\win32\mexopts\lccengmatopts.bat'];
mex('-f', optsfile, 'engwindemo.c');
```

Verify that the build worked by looking in your current working directory for the file engwindemo.exe:

```
dir engwindemo.exe
```

To run the demo from MATLAB, make sure your current working directory is set to the one in which you built the executable file, and then type:

```
!engwindemo
```

Example — Building an Engine Application on UNIX Systems

MATLAB software provides demonstration programs written in C and C++ that you can use to verify the build process on your computer. The demo files for UNIX systems are engdemo.c and engdemo.cpp.

Copy one of the demonstration programs, for example, engdemo.c, to your current working directory:

```
demofile = [matlabroot '/extern/examples/eng_mat/engdemo.c'];
copyfile(demofile, '.');
```

Build the executable file using the ANSI compiler for engine stand alone programs and the options file engopts.sh:

```
optsfile = [matlabroot '/bin/engopts.sh'];
mex('-f', optsfile, 'engdemo.c');
```

Verify that the build worked by looking in your current working directory for the file engdemo:

```
dir engdemo
```

To run the demo in MATLAB, make sure your current working directory is set to the one in which you built the executable file, and then type:

```
!engdemo
```

7

Using Sun Java Classes in MATLAB Software

Product Overview

In this section...
"Sun Java Interface Is Integral to MATLAB Software" on page 7-2
"Benefits of the MATLAB Java Interface" on page 7-2
"Who Should Use the MATLAB Java Interface" on page 7-2
"To Learn More About Java Programming Language" on page 7-3
"Platform Support for JVM Software" on page 7-3
"Using a Different Version of JVM Software" on page 7-4

Sun Java Interface Is Integral to MATLAB Software

Every installation of MATLAB software includes Java Virtual Machine (JVM) software, so that you can use the Java interpreter via MATLAB commands, and you can create and run programs that create and access Java objects. For information on the MATLAB installation, see the MATLAB installation documentation for your platform.

Benefits of the MATLAB Java Interface

The MATLAB Java interface enables you to:

- Access Java API (application programming interface) class packages that support essential activities such as I/O and networking. For example, the URL class provides convenient access to resources on the Internet.

- Access third-party Java classes

- Easily construct Java objects in MATLAB workspace

- Call Java object methods, using either Java or MATLAB syntax

- Pass data between MATLAB variables and Java objects

Who Should Use the MATLAB Java Interface

The MATLAB Java interface is intended for all MATLAB users who want to take advantage of the special capabilities of the Java programming language.

For example:

- You need to access, from MATLAB, the capabilities of available Java classes.
- You are familiar with object-oriented programming in Java or in another language, such as C++.
- You are familiar with *Object-Oriented Programming*, or with MATLAB MEX-files.

To Learn More About Java Programming Language

For a complete description of the Java language and for guidance in object-oriented software design and programming, you'll need to consult outside resources. For example, these recently published books may be helpful:

- *Java in a Nutshell* (Fourth Edition), by David Flanagan
- *Teach Yourself Java in 21 Days*, by Lemay and Perkins

Another place to find information is the JavaSoft Web site.

```
http://www.javasoft.com
```

For other suggestions on object-oriented programming resources, see:

- *Object-Oriented Software Construction*, by Bertrand Meyer
- *Object-Oriented Analysis and Design with Applications*, by Grady Booch, Robert A. Maksimchuk, Michael W. Engel, and Alan Brown

Platform Support for JVM Software

To find out which version of JVM software is used by MATLAB on your platform, type the following at the MATLAB prompt:

```
version -java
```

Using a Different Version of JVM Software

MATLAB ships with one specific version of JVM software and uses this version by default with the MATLAB interface to the Java language. This section describes how to download and select a version other than the default.

Note MATLAB is only fully supported on the JVM software that it ships with. Some components might not work properly under a different version of the JVM software. For example, calling functions in a dynamically linked library that was created with a different JVM software version than that used by MATLAB might cause a segmentation violation error message.

To change the JVM software version that MATLAB uses, follow these steps:

1 "Download the JVM Software Version You Want to Use" on page 7-4.

2 "Locate the Root of the Run-time Path for this Version" on page 7-4.

3 "Set the MATLAB_JAVA Environment Variable to this Path" on page 7-5.

To verify that MATLAB is using the correct version of the JVM software, type the version -java command.

Download the JVM Software Version You Want to Use

You can download JVM software from the Web site .

Locate the Root of the Run-time Path for this Version

To get MATLAB to use the version you have just downloaded, you must first find the root of the run-time path for this JVM software version, and then set the MATLAB_JAVA environment variable to that path. To locate the JVM run-time path, find the directory in the Java installation tree that is one level up from the directory containing the file rt.jar. This may be a subdirectory of the main Sun JDK™ install directory. (If you cannot find rt.jar, look for the file classes.zip.)

For example, if the JDK software is installed in `D:\jdk1.2.1` on a Microsoft Windows system and the `rt.jar` file is in `D:\jdk1.2.1\jre\lib`, set `MATLAB_JAVA` to the directory one level up from that: `D:\jdk1.2.1\jre`.

On a UNIX[21] system, if the JDE software is installed in `/usr/openv/java/jre/lib` and the `rt.jar` is in `/usr/openv/java/jre/lib`, set `MATLAB_JAVA` to the path `/usr/openv/java/jre`.

Set the MATLAB_JAVA Environment Variable to this Path

The way you set or modify the value of the `MATLAB_JAVA` variable depends on which platform you are running MATLAB on.

Windows XP Operating System.

1 Click **Settings** in the **Start** Menu.

2 Choose **Control Panel**.

3 Click **System**.

4 Choose the **Advanced** tab, and then click the **Environment Variables** button.

5 You now can set (or add) the `MATLAB_JAVA` system environment variable to the path of your JVM software.

UNIX or Linux[22] Operating Systems.

```
setenv MATLAB_JAVA <path to JVM>
```

---

21. UNIX is a registered trademark of The Open Group in the United States and other countries.

22. Linux is a registered trademark of Linus Torvalds.

Bringing Java Classes and Methods into MATLAB Workspace

Introduction

You can draw from an extensive collection of existing Sun Java classes or create your own class definitions to use with MATLAB software. This section explains how to go about finding the class definitions that you need or how to create classes of your own design. Once you have the classes you need, defined in either individual `.class` files, packages, or Java Archive (JAR) files, you can make them available in the MATLAB workspace. This section also describes how to specify the native method libraries used by Java code.

Sources of Java Classes

Following are Java class sources that you can use in the MATLAB workspace:

- Java built-in classes — general-purpose class packages, such as `java.util`, included in the Java language. See your Java language documentation for descriptions of these packages.

- Third-party classes — packages of special-purpose Java classes.

- User-defined classes — Java classes or subclasses of existing classes that you define. You need to use a Java language development environment to do this, as explained in the following section.

Defining New Java Classes

To define new Java classes and subclasses of existing classes, you must use a Java language development environment external to MATLAB software. See `Technical Note 1601` for information on supported versions of the Java Development Kit (JDK) software. You can download the JDK from the Sun Microsystems™ Web site, (`http://java.sun.com/j2se/`). The Sun site also provides documentation for the Java language and classes that you need for development.

After you create class definitions in `.java` files, use your Java compiler to produce `.class` files from them. The next step is to make the class definitions in those `.class` files available for you to use in MATLAB.

The Java Class Path

MATLAB loads Java class definitions from files that are on the Java *class path*. The class path is a series of file and directory specifications that MATLAB software uses to locate class definitions. When loading a particular Java class, MATLAB searches files and directories in the order they occur on the class path until a file is found that contains that class definition. The search ends when the first definition is found.

The Java class path consists of two segments: the *static path* and the *dynamic path*. MATLAB loads the static path at startup. If you change the path you must restart MATLAB. You can load and modify the dynamic path at any time using MATLAB functions. MATLAB always searches the static path before the dynamic path.

Note Java classes on the static path should not have dependencies on classes on the dynamic path.

You can view these two path segments using the `javaclasspath` function:

```
javaclasspath

        STATIC JAVA PATH

    D:\SysO\Java\util.jar
    D:\SysO\Java\widgets.jar
    D:\SysO\Java\beans.jar
                .
                .

        DYNAMIC JAVA PATH

    C:\Work\Java\ClassFiles
    C:\Work\Java\mywidgets.jar
                .
                .
```

You probably want to use both the static and dynamic paths:

- Put the Java class definitions that are more stable on the static class path. Classes defined on the static path load somewhat faster than those on the dynamic path.

- Put the Java class definitions that you are likely to modify on the dynamic class path. You can make changes to the class definitions on this path without restarting MATLAB.

The Static Path

MATLAB loads the static class path from the `classpath.txt` file at the start of each session. The static path offers better class loading performance than the dynamic path. However, to modify the static path, you need to edit `classpath.txt`, and then restart MATLAB.

Finding and Editing classpath.txt. The default `classpath.txt` file resides in the `toolbox\local` subdirectory of your MATLAB root directory `matlabroot`:

```
[matlabroot '\toolbox\local\classpath.txt']
ans =
    \\sys07\matlab\toolbox\local\classpath.txt
```

To make changes in the static path that affect all users who share this same MATLAB root directory, edit this file in `toolbox\local`. If you want to make changes that do not affect anyone else, copy `classpath.txt` to your own startup directory and edit the file there. When MATLAB starts up, it looks for `classpath.txt` first in your startup directory, and then in the default location. It uses the first file it finds.

To see which `classpath.txt` file is currently being used by your MATLAB environment, use the `which` function:

```
which classpath.txt
```

To edit either the default file or the copy in your own directory, type:

```
edit classpath.txt
```

Note MATLAB reads `classpath.txt` only at startup. If you edit `classpath.txt` or change your `.class` files while MATLAB is running, you must restart MATLAB to put those changes into effect.

Special Symbols in classpath.txt. You can designate special tokens or macros in the `classpath.txt` file using a leading dollar sign, (e.g., `$matlabroot` or `$jre_home`). However, this can cause problems if you use this sign in any of your class directory paths. For example, the following path string does not correctly represent the path to a directory named `hello$world`:

```
d:\applications\hello$world
```

You must use two consecutive dollar signs in `classpath.txt` to represent a single `$` character. So, to correctly specify the directory path shown above, you need to use the following text:

```
d:\applications\hello$$world
```

The Dynamic Path

The dynamic class path can be loaded any time during a MATLAB software session using the `javaclasspath` function. You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear` with the keyword `java`) without restarting MATLAB. See the Java function reference pages for more information on how to use these functions.

The functions `javaaddpath` and `javaclasspath(dpath)` add entries to the dynamic class path. To avoid the possibility that the new path contains a class or package with the same name as an existing class or package, MATLAB clears all existing global variables and variables in the workspace.

Although the dynamic path offers more flexibility in changing the path, Java classes on the dynamic path may load more slowly than those on the static path.

Making Java Classes Available in MATLAB Workspace

To make your third-party and user-defined Java classes available in the MATLAB workspace, place them on either the static or dynamic Java class path, as described in the previous section, "The Java Class Path" on page 7-7.

- For classes you want on the static path, edit the `classpath.txt` file.

- For classes you want on the dynamic path, use either the `javaclasspath` or the `javaaddpath` functions.

Making Individual (Unpackaged) Classes Available

To make individual classes (classes that are not part of a package) available in MATLAB, specify the full path to the directory you want to use for the `.class` file(s).

For example, to make available your compiled Java classes in the file `d:\work\javaclasses\test.class`, add the following entry to the static or dynamic class path:

```
d:\work\javaclasses
```

To put this directory on the static class path, add the above line to the default copy (in `toolbox\local`) or your own local copy of `classpath.txt`. See "Finding and Editing classpath.txt" on page 7-8.

To put this on the dynamic class path, use the following command:

```
javaaddpath d:\work\javaclasses
```

Making Entire Packages Available

To access one or more classes belonging to a package, you need to make the entire package available to MATLAB. To do this, specify the full path to the *parent directory of the highest level directory* of the package path. This directory is the first component in the package name.

For example, if your Java class package `com.mw.tbx.ini` has its classes in directory `d:\work\com\mw\tbx\ini`, add the following directory to your static or dynamic class path:

```
d:\work
```

Making Classes in a JAR File Available

You can use the `jar` (Java Archive) tool to create a JAR file, containing multiple Java classes and packages in a compressed ZIP format. For information on `jar` and JAR files, consult your Java development documentation or the JavaSoft Web site `http://www.javasoft.com`. See also "To Learn More About Java Programming Language" on page 7-3.

To make the contents of a JAR file available for use in MATLAB, specify the full path, *including full filename*, for the JAR file.

Note The `classpath.txt` requirement for JAR files is different than that for `.class` files and packages, for which you do not specify any filename.

For example, to make available the JAR file `e:\java\classes\utilpkg.jar`, add the following file specification to your static or dynamic class path:

```
e:\java\classes\utilpkg.jar
```

Loading Java Class Definitions

Normally, MATLAB software loads a Java class automatically when your code first uses it, (for example, when you call its constructor). However, there is one exception you should be aware of.

When you use the `which` function on methods defined by Java classes, the function only acts on the classes currently *loaded* into the MATLAB workspace. In contrast, `which` always operates on MATLAB classes, whether or not they are loaded.

Determining Which Classes Are Loaded

At any time during a MATLAB software session, you can obtain a listing of all the Java classes that are currently loaded. To do so, use the `inmem` function as follows:

```
[M,X,J] = inmem
```

This function returns the list of Java classes in the output argument `J`. (It also returns the names of all currently loaded M-files in `M`, and the names of all currently loaded MEX-files in `X`.)

Here's a sample of output from the `inmem` function:

```
[m,x,j] = inmem;
```

MATLAB displays:

```
j =
 'java.util.Date'
 'com.mathworks.ide.desktop.MLDesktop'
```

Simplifying Java Class Names

Your MATLAB commands can refer to any Java class by its fully qualified name, which includes its package name. For example, the following are fully qualified names:

• `java.lang.String`

- `java.util.Enumeration`

A fully qualified name can be rather long, making commands and functions, such as constructors, cumbersome to edit and to read. You can refer to classes by the class name alone (without a package name) if you first import the fully qualified name into MATLAB.

The `import` command has the following forms:

```
import pkg_name.*              % Import all classes in package
import pkg_name1.* pkg_name2.* % Import multiple packages
import class_name              % Import one class
import                         % Display current import list
L = import                     % Return current import list
```

MATLAB adds all classes that you import to a list called the *import list*. You can see what classes are on that list by typing `import`, without any arguments. Your code can refer to any class on the list by class name alone.

When called from a function, `import` adds the specified classes to the import list in effect for that function. When invoked at the command prompt, `import` uses the base import list for your MATLAB software environment.

For example, suppose a function contains the following statements:

```
import java.lang.String
import java.util.* java.awt.*
import java.util.Enumeration
```

Any code that follows these `import` statements can refer to the `String`, `Frame`, and `Enumeration` classes without using the package names. For example:

```
str = String('hello');    % Create java.lang.String object
frm = Frame;              % Create java.awt.Frame object
methods Enumeration       % List java.util.Enumeration methods
```

To `clear` the list of imported Java classes, type:

```
clear import
```

Locating Native Method Libraries

Java classes can dynamically load native methods using the Java method `java.lang.System.loadLibrary("LibFile")`. In order for the Sun JVM software to locate the specified library file, the directory containing it must be on the Java Library Path. This path is established when the MATLAB software launches the JVM software at startup, and is based on the contents of the file:

    matlabroot/toolbox/local/librarypath.txt

You can augment the search path for native method libraries by editing the `librarypath.txt` file. Follow these guidelines when editing this file:

- Specify each new directory on a line by itself.

- Specify only the directory names, not the names of the DLL files. The `loadLibrary` call does this for you.

- To simplify the specification of directories in cross-platform environments, use any of these macros: `$matlabroot`, `$arch`, and `$jre_home`.

Java Classes Contained in a JAR File

You can access Java classes that are contained in a JAR file once you have added the JAR file to either the static or dynamic class path. See "The Java Class Path" on page 7-7 for more information on how MATLAB software uses the Java class path.

For example, suppose you have a file, `myArchive.jar`, in a directory called `work` in your MATLAB root directory. You can construct the path to this file using the `matlabroot` command:

    `[matlabroot '/work/myArchive.jar']`

Add the JAR file to your dynamic class path using the `javaaddpath` function (`fullfile` adds the platform-correct directory separators):

    `javaaddpath(fullfile(matlabroot,'work','myArchive.jar'))`

You can now call the public methods in the JAR file.

Creating and Using Java Objects

In this section...
"Overview" on page 7-15
"Constructing Java Objects" on page 7-15
"Concatenating Java Objects" on page 7-18
"Saving and Loading Java Objects to MAT-Files" on page 7-19
"Finding the Public Data Fields of an Object" on page 7-20
"Accessing Private and Public Data" on page 7-20
"Determining the Class of an Object" on page 7-22

Overview

You create a Sun Java object in the MATLAB workspace by calling one of the constructors of that class. You then use commands and programming statements to perform operations on these objects. You can also save your Java objects to a MAT-file and, in subsequent sessions, reload them into MATLAB.

Constructing Java Objects

You construct Java objects in the MATLAB workspace by calling the Java class constructor, which has the same name as the class. For example, the following constructor creates a `myDate` object:

```
myDate = java.util.Date
```

MATLAB displays information similar to:

```
myDate =
Thu Aug 23 12:58:54 EDT 2007
```

All of the programming examples in this chapter contain Java object constructors. For example, the code in the Example — Reading a URL creates a `java.net.URL` object with the constructor:

```
url = java.net.URL(...
```

```
'http://archive.ncsa.uiuc.edu/demoweb/')
```

Using the javaObject Function

Under certain circumstances, you may need to use the javaObject function to construct a Java object. The following syntax invokes the Java constructor for class, class_name, with the argument list that matches x1,...,xn, and returns a new object, J.

```
J = javaObject('class_name',x1,...,xn);
```

For example, to construct and return a Java object of class java.lang.String, type:

```
strObj = javaObject('java.lang.String','hello');
```

With the javaObject function you can:

- Use classes that have names that exceed the maximum length of a MATLAB identifier. (Call the namelengthmax function to obtain the maximum identifier length.)

- Specify the class for an object at run-time, for example, as input from an application user

The default MATLAB constructor syntax requires that no segment of the input class name be longer than namelengthmax characters. (A *class name segment* is any portion of the class name before, between, or after a dot. For example, there are three segments in class, java.lang.String.) Any class name segment that exceeds namelengthmax characters is truncated by MATLAB. In the rare case where you need to use a class name of this length, you must use javaObject to instantiate the class.

The javaObject function also allows you to specify the Java class for the object being constructed at run-time. In this situation, you call javaObject with a string variable in place of the class name argument.

```
class = 'java.lang.String';
text = 'hello';
strObj = javaObject(class, text);
```

In the usual case, when the class to instantiate is known at development time, it is more convenient to use the MATLAB constructor syntax. For example, to create a `java.lang.String` object, type:

```
strObj = java.lang.String('hello');
```

Note Typically, you do not need to use `javaObject`. The default MATLAB syntax for instantiating a Java class is somewhat simpler and is preferable for most applications. Use `javaObject` primarily for the two cases described above.

Java Objects Are References in MATLAB Software Applications

In MATLAB, Java objects are *references* and do not adhere to MATLAB copy-on-assignment and pass-by-value rules. For example:

```
myDate = java.util.Date;
setHours(myDate, 10)
newDate = myDate;
```

In this example, the variable `newDate` is a reference to `myDate`, not a copy of the object. Any change to the object referenced by `newDate` also changes the object at `myDate`. This happens if the object is changed by MATLAB code or by Java code.

The following example shows that `myDate` and `newDate` are references to the same object. When you change the hour via one reference (`newDate`), the change is reflected through the other reference (`myDate`), as well.

```
setHours(newDate, 8)
myDate.getHours
```

MATLAB displays:

```
ans =
     8
```

Concatenating Java Objects

You can concatenate Java objects in the same way that you concatenate native MATLAB types. You use either the cat function or the [ ] operators to tell MATLAB software to assemble the enclosed objects into a single object.

Concatenating Objects of the Same Class

If all of the objects being operated on are of the same Java class, the concatenation of those objects produces an array of objects from the same class.

In the following example, the cat function concatenates two objects of the class java.awt.Integer. The class of the result is also java.awt.Integer.

```
value1 = java.lang.Integer(88);
value2 = java.lang.Integer(45);
cat(1, value1, value2)
```

MATLAB displays:

```
ans =
java.lang.Integer[]:
    [88]
    [45]
```

Concatenating Objects of Unlike Classes

When you concatenate objects of unlike classes, MATLAB finds one class from which all of the input objects inherit, and makes the output an instance of this class. MATLAB selects the lowest common parent in the Java class hierarchy as the output class.

For example, concatenating objects of java.lang.Byte, java.lang.Integer, and java.lang.Double creates an object of java.lang.Number, since this is the common parent to the three input classes.

```
byte = java.lang.Byte(127);
integer = java.lang.Integer(52);
double = java.lang.Double(7.8);
[byte; integer; double]
```

MATLAB displays:

```
ans =
java.lang.Number[]:
    [   127]
    [    52]
    [7.8000]
```

If there is no common, lower level parent, then the resultant class is
`java.lang.Object`, which is the root of the entire Java class hierarchy.

```
byte = java.lang.Byte(127);
point = java.awt.Point(24,127);
[byte; point]
```

MATLAB displays:

```
ans =
java.lang.Object[]:
    [               127]
    [1x1 java.awt.Point]
```

Saving and Loading Java Objects to MAT-Files

Use the `save` function to save a Java object to a MAT-file. Use the `load`
function to load it back into MATLAB from that MAT-file. To save a Java
object to a MAT-file, and to load the object from the MAT-file, make sure that
the object and its class meet all of the following criteria:

- The class implements the `Serializable` interface (part of the Java API),
  either directly or by inheriting it from a parent class. Any embedded or
  otherwise referenced objects must also implement `Serializable`.

- The definition of the class is not changed between saving and loading the
  object. Any change to the data fields or methods of a class prevents the
  loading (deserialization) of an object that was constructed with the old
  class definition.

- Either the class does not have any transient data fields, or the values in
  transient data fields of the object to be saved are not significant. Values in
  transient data fields are never saved with the object.

If you define your own Java classes, or subclasses of existing classes, you can follow the criteria above to enable objects of the class to be saved and loaded in MATLAB. For details on defining classes to support serialization, consult your Java development documentation. (See also "To Learn More About Java Programming Language" on page 7-3.)

Finding the Public Data Fields of an Object

To list the public fields that belong to a Java object, use the `fieldnames` function, which takes either of these forms.

```
names = fieldnames(obj)
names = fieldnames(obj,'-full')
```

Calling `fieldnames` without `-full` returns the names of all the data fields (including inherited) on the object. With the `-full` qualifier, `fieldnames` returns the full description of the data fields defined for the object, including type, attributes, and inheritance information.

For example, create an `Integer` object with the command:

```
value = java.lang.Integer(0);
```

To see a full description of the data fields of `value`, type:

```
fieldnames(value, '-full')
```

MATLAB displays:

```
ans =
    'static final int MIN_VALUE'
    'static final int MAX_VALUE'
    'static final java.lang.Class TYPE'
    'static final int SIZE'
```

Accessing Private and Public Data

Java API classes provide accessor methods you can use to read from and, where allowed, to modify *private* data fields. These are sometimes referred to as *get* and *set* methods, respectively.

Some Java classes have *public* data fields, which your code can read or modify directly. To access these fields, use the syntax `object.field`.

Examples

The `java.awt.Frame` class provides an example of access to both private and public data fields. This class has the read accessor method `getSize`, which returns a `java.awt.Dimension` object. The `Dimension` object has data fields `height` and `width`, which are public and therefore directly accessible. For example, to access this data, type:

```
frame = java.awt.Frame;
frameDim = getSize(frame);
height = frameDim.height;
frameDim.width = 42;
```

The programming examples in this chapter also contain calls to data field accessors. For instance, the sample code for "Example — Finding an Internet Protocol Address" on page 7-74 uses calls to accessors on a `java.net.InetAddress` object.

```
hostname = address.getHostName;
ipaddress = address.getHostAddress;
```

Accessing Data from a Static Field

In a Java language program, a *static data field* is a field that applies to an entire class of objects. Static fields are most commonly accessed in relation to the class name itself. For example, the following code accesses the `TYPE` field of the `Integer` class by referring to it in relation to the package and class names, `java.lang.Integer`, rather than an object instance.

```
thisType = java.lang.Integer(0).TYPE;
```

In MATLAB, you can use that same syntax. Or you can refer to the `TYPE` field in relation to an instance of the class. The following example creates an instance of `java.lang.Integer` called `value`, and then accesses the `TYPE` field using the name `value` rather than the package and class names.

```
value = java.lang.Integer(0);
thatType = value.TYPE
```

MATLAB displays:

```
thatType =
int
```

Assigning to a Static Field

You can assign values to static fields by using a static `set` method of the class, or by making the assignment in reference to an instance of the class. For more information, see "Accessing Data from a Static Field" on page 7-21. You can assign `value` to the field `staticFieldName` in the following example by referring to this field in reference to an instance of the class.

```
objectName = java.className;
objectName.staticFieldName = value;
```

Note MATLAB does not allow assignment to static fields using the class name itself.

Determining the Class of an Object

To find the class of a Java object, use the query form of the `class` function. After execution of the following example, `myClass` contains the name of the package and class that the object `value` instantiates.

```
value = java.lang.Integer(0);
myClass = class(value)
```

MATLAB displays:

```
myClass =
java.lang.Integer
```

Because this form of `class` also works on MATLAB objects, it does not, in itself, tell you whether it is a Java class. To determine the type of class, use the `isjava` function, which has the form:

```
x = isjava(obj)
```

`isjava` returns `1` if `obj` is a Java object, and `0` if it is not. For example, type:

```
isjava(value)
```

MATLAB displays:

```
ans =
     1
```

To find out if an object is an instance of a specified class, use the `isa` function, which has the form:

```
x = isa(obj, 'class_name')
```

`isa` returns `1` if `obj` is an instance of the class named `'class_name'`, and `0` if it is not. Note that `'class_name'` can be a MATLAB built-in or user-defined class, as well as a Java class. For example, type:

```
isa(value, 'java.lang.Integer')
```

MATLAB displays:

```
ans =
     1
```

Invoking Methods on Java Objects

In this section...

Using Java and MATLAB Calling Syntax

To call methods on Sun Java objects, you can use the Java syntax:

```
object.method(arg1,...,argn)
```

In the following example, `myDate` is a `java.util.Date` object, and `getHours` and `setHours` are methods of that object.

```
myDate = java.util.Date;
myDate.setHours(3)
myDate.getHours
```

The MATLAB software displays:

```
ans =
     3
```

Alternatively, you can call Java object (nonstatic) methods with the MATLAB syntax:

```
method(object, arg1,...,argn)
```

Using MATLAB syntax:

```
getHours(myDate)
```

MATLAB displays:

```
ans =
     3
```

All of the programming examples in this chapter contain invocations of Java object methods. For example, the code for "Example — Reading a URL" on page 7-71 contains a call, using MATLAB syntax, to the `openStream` method on a `java.net.URL` object, `url`.

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

In another example, the code for "Example — Creating and Using a Phone Book" on page 7-76 contains a call, using Java syntax, to the `load` method on a `java.utils.Properties` object, `pb_htable`.

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

Using the javaMethod Function on Nonstatic Methods

Under certain circumstances, you may need to use the `javaMethod` function to call a Java method. The following syntax invokes the method, `method_name`, on Java object J with the argument list that matches `x1,...,xn`. This returns the value X.

```
X = javaMethod('method_name',J,x1,...,xn);
```

For example, to call the `startsWith` method on a `java.lang.String` object passing one argument, use:

```
gAddress = java.lang.String('Four score and seven years ago');
str = java.lang.String('Four score');
javaMethod('startsWith', gAddress, str)
ans =
     1
```

Using the `javaMethod` function enables you to:

- Use methods that have names that exceed the maximum length of a
  MATLAB identifier. (Call the `namelengthmax` function to obtain the
  maximum identifier length.)

- Specify the method you want to invoke at run-time, for example, as input
  from an application user.

The only way to invoke a method whose name is longer than `namelengthmax`
characters is to use `javaMethod`. The Java and MATLAB calling syntax does
not accept method names of this length.

With `javaMethod`, you can also specify the method to be invoked at run time.
In this situation, your code calls `javaMethod` with a string variable in place
of the `method_name` argument. When you use `javaMethod` to invoke a static
method, you can also use a string variable in place of the class name argument.

Note Typically, you do not need to use `javaMethod`. The default MATLAB
syntax for invoking a Java method is somewhat simpler and is preferable for
most applications. Use `javaMethod` primarily for the two cases described
above.

Invoking Static Methods on Java Classes

To invoke a `static` method on a Java class, use the Java syntax:

```
class.method(arg1,...,argn)
```

For example, to call the `isNaN` static method on the `java.lang.Double` class,
type:

```
java.lang.Double.isNaN(2.2)
```

Alternatively, you can apply static method names to instances of a class.
In this example, the `isNaN` static method is referenced in relation to the
`dblObject` instance of the `java.lang.Double` class.

```
dblObject = java.lang.Double(2.2);
dblObject.isNaN
ans =
     0
```

Using the javaMethod Function on Static Methods

You can use the `javaMethod` function to call static methods.

The following syntax invokes the static method, `method_name`, in class, `class_name`, with the argument list that matches `x1,...,xn`. This returns the value `X`.

```
X = javaMethod('method_name','class_name',x1,...,xn);
```

For example, to call the static `isNaN` method of the `java.lang.Double` class on a double value of 2.2, type:

```
javaMethod('isNaN','java.lang.Double',2.2);
```

Using the `javaMethod` function to call static methods enables you to:

- Use methods that have names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)
- Specify method and class names at run-time, for example, as input from an application user.

Obtaining Information About Methods

MATLAB software offers several functions to help obtain information related to the Java methods you are working with. You can request a list of all of the methods that are implemented by any class. The list may be accompanied by other method information such as argument types and exceptions. You can also request a listing of every Java class that you loaded into MATLAB that implements a specified method.

Methodsview: Displaying a Listing of Java Methods

If you want to know what methods are implemented by a particular Java (or MATLAB) class, use the `methodsview` function. Specify the class name (along with its package name, for Java classes) in the command line. If you have imported the package that defines this class, then the class name alone suffices.

The following command lists information on all methods in the `java.awt.MenuItem` class. Type:

```
methodsview java.awt.MenuItem
```

A new window appears, listing one row of information for each method in the class.
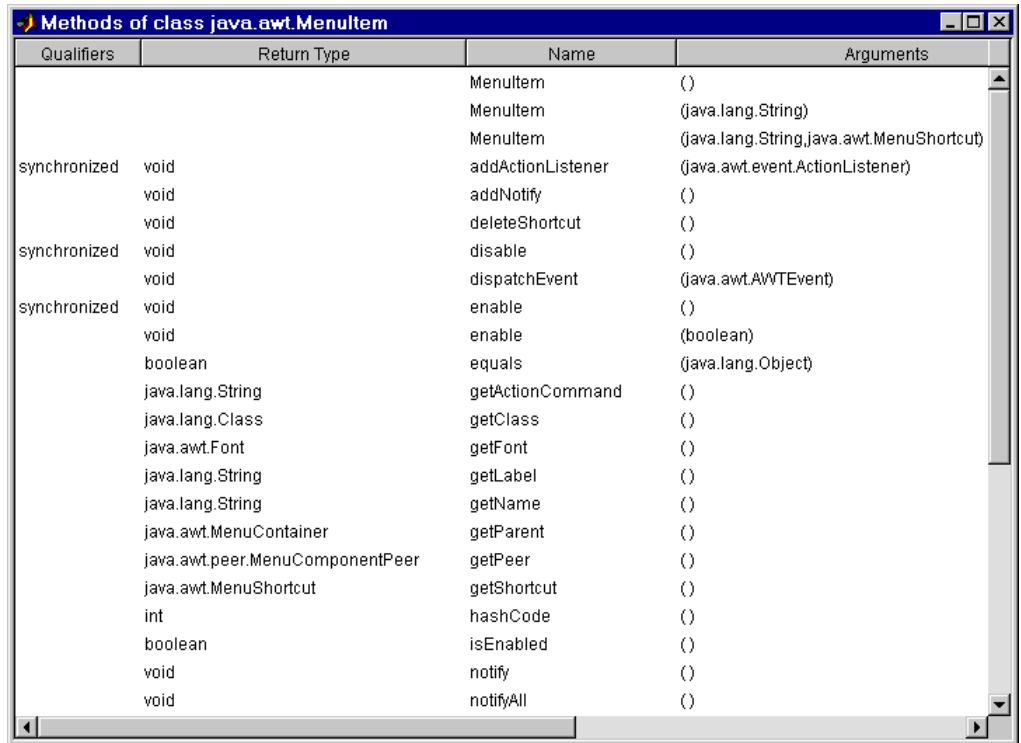
Each row in the window displays up to six fields of information describing the method. The following table lists the fields displayed in the methodsview window along with a description and examples of each field type.

Fields Displayed in the Methodsview Window

Field Name	Description	Examples
Qualifiers	Method type qualifiers	abstract, synchronized

Fields Displayed in the Methodsview Window (Continued)

Field Name	Description	Examples
Return Type	Type returned by the method	`void`, `java.lang.String`
Name	Method name	`addActionListener`, `dispatchEvent`
Arguments	Types of arguments passed to method	`boolean`, `java.lang.Object`
Other	Other relevant information	`throws java.io.IOException`
Inherited From	Parent of the specified class	`java.awt.MenuComponent`

Using the Methods Function on Java Classes

The `methods` function returns information on methods of MATLAB and Java classes. You can use any of the following forms of this command.

```
methods class_name
methods class_name -full
n = methods('class_name')
n = methods('class_name','-full')
```

Use `methods` without the `'-full'` qualifier to return the names of all the methods (including inherited methods) of the class. Names of overloaded methods are listed only once.

With the `'-full'` qualifier, `methods` returns a listing of the method names (including inherited methods) along with attributes, argument lists, and inheritance information on each. Each overloaded method is listed separately.

For example, display a full description of all methods of the `java.awt.Dimension` object.

```
methods java.awt.Dimension -full

Methods for class java.awt.Dimension:
```

```
Dimension()
Dimension(java.awt.Dimension)
Dimension(int,int)
java.lang.Class getClass()  % Inherited from java.lang.Object
int hashCode()  % Inherited from java.lang.Object
boolean equals(java.lang.Object)
java.lang.String toString()
void notify()  % Inherited from java.lang.Object
void notifyAll()  % Inherited from java.lang.Object
void wait(long) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long,int) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait() throws java.lang.InterruptedException
   % Inherited from java.lang.Object
java.awt.Dimension getSize()
void setSize(java.awt.Dimension)
void setSize(int,int)
```

Determining What Classes Define a Method

You can use the which function to display the fully qualified name (package and class name) of a method implemented by a *loaded* Java class. With the -all qualifier, the which function finds all classes with a method of the name specified.

Suppose, for example, that you want to find the package and class name for the concat method, with the String class currently loaded. Use the command:

```
which concat
java.lang.String.concat  % String method
```

If the java.lang.String class has not been loaded, the same which command would give the output:

```
which concat
concat not found.
```

If you use which -all for the method equals, with the String and java.awt.Frame classes loaded, you see the following display.

```
which -all equals
java.lang.String.equals                          % String method
java.awt.Frame.equals                            % Frame method
com.mathworks.ide.desktop.MLDesktop.equals   % MLDesktop method
```

The `which` function operates differently on Java classes than it does on MATLAB classes. MATLAB classes are always displayed by `which`, whether or not they are loaded. This is not true for Java classes. You can find out which Java classes are currently loaded by using the command `[m,x,j]=inmem`, described in "Determining Which Classes Are Loaded" on page 7-12.

For a description of how Java classes are loaded, see "Making Java Classes Available in MATLAB Workspace" on page 7-10.

Java Methods That Affect MATLAB Commands

MATLAB commands that operate on Java objects and arrays make use of the methods that are implemented within, or inherited by, these objects' classes. There are some MATLAB commands that you can alter somewhat in behavior by changing the Java methods that they rely on.

Changing the Effect of disp and display

You can use the `disp` function to display the value of a variable or an expression in MATLAB. Terminating a command line without a semicolon also calls the `disp` function. You can also use `disp` to display a Java object in MATLAB.

When `disp` operates on a Java object, MATLAB formats the output using the `toString` method of the class to which the object belongs. If the class does not implement this method, then an inherited `toString` method is used. If no intermediate ancestor classes define this method, it uses the `toString` method defined by the `java.lang.Object` class. You can override inherited `toString` methods in classes that you create by implementing such a method within your class definition. In this way, you can change the way MATLAB displays information regarding the objects of the class.

Changing the Effect of isequal

The MATLAB isequal function compares two or more arrays for equality in type, size, and contents. This function can also be used to test Java objects for equality.

When you compare two Java objects using isequal, MATLAB performs the comparison using the Java method, equals. MATLAB first determines the class of the objects specified in the command, and then uses the equals method implemented by that class. If it is not implemented in this class, then an inherited equals method is used. This is the equals method defined by the java.lang.Object class if no intermediate ancestor classes define this method.

You can override inherited equals methods in classes that you create by implementing such a method within your class definition. In this way, you can change the way MATLAB performs comparison of the members of this class.

Changing the Effect of double and char

You can also define your own Java methods toDouble and toChar to change the output of the MATLAB double and char functions. For more information, see "Converting to the MATLAB double Type" on page 7-65 and "Converting to the MATLAB char Type" on page 7-66.

How MATLAB Software Handles Undefined Methods

If your MATLAB command invokes a nonexistent method on a Java object, MATLAB looks for a function with the same name. If MATLAB finds a function of that name, it attempts to invoke it. If MATLAB does not find a function with that name, it displays a message stating that it cannot find a method by that name for the class.

For example, MATLAB has a function named size, and the Java API java.awt.Frame class also has a size method. If you call size on a Frame object, the size method defined by java.awt.Frame is executed. However, if you call size on an object of java.lang.String, MATLAB does not find a size method for this class. It executes the MATLAB size function instead.

```
string = java.lang.String('hello');
size(string)
ans =
     1     1
```

Note When you define a Java class for use in MATLAB, avoid giving any of its methods the same name as a MATLAB function.

How MATLAB Software Handles Java Exceptions

If invoking a Java method or constructor throws an exception, MATLAB catches the exception and transforms it into a MATLAB error message. MATLAB puts the text of the Java error message into its own error message. Receiving an error from a Java method or constructor has the same appearance as receiving an error from an M-file.

Method Execution in MATLAB Software

When calling a main method from MATLAB, the method returns as soon as it executes its last statement, even if the method creates a thread that is still executing. In other environments, the main method does not return until the thread completes execution.

You, therefore, need to be cautious when calling main methods from MATLAB, particularly main methods that launch GUIs. main methods are usually written assuming they are the entry point to application code. When called from MATLAB this is not the case, and the fact that other Java GUI code might be already running can lead to problems.

Working with Java Arrays

In this section...

Introduction

You can pass singular Sun Java objects to and from methods or you may pass them in an array, providing the method expects them in that form. This array must either be a Java array (returned from another method call or created within the MATLAB software) or, under certain circumstances, a MATLAB cell array. This section describes how to create and manipulate Java arrays in MATLAB. Later sections will describe how to use MATLAB cell arrays in calls to Java methods.

Note The term *dimension* here 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 being one-dimensional, as its individual elements can be indexed into using only one array subscript.

How MATLAB Software Represents the Java Array

The term *Java array* refers to any array of Java objects returned from a call to a Java class constructor or method. You may also construct a Java array within MATLAB using the javaArray function. The structure of a Java array is significantly different from that of a MATLAB matrix or array. MATLAB

hides these differences whenever possible, allowing you to operate on the arrays using the usual MATLAB command syntax. Just the same, it may be helpful to keep the following differences in mind as you work with Java arrays.

Representing More Than One Dimension

An array in the Java language is strictly a one-dimensional structure because it is measured only in length. If you want to work with a two-dimensional array, you can create an equivalent structure using an array of arrays. To add further dimensions, you add more levels to the array, making it an array of arrays of arrays, and so on. You may want to use such multilevel arrays when working in MATLAB as it is a matrix and array-based programming language.

MATLAB makes it easy for you to work with multilevel Java arrays by treating them like the matrices and multidimensional arrays that are a part of the language itself. You access elements of an array of arrays using the same MATLAB syntax that you use if you are handling a matrix. If you add more levels to the array, MATLAB can access and operate on the structure as if it is a multidimensional MATLAB array.

The left side of the following figure shows Java arrays of one, two, and three dimensions. To the right of each is the way the same array is represented to you in MATLAB. Note that single-dimension arrays are represented as a column vector.

Array Access from Java   Array Access from MATLAB
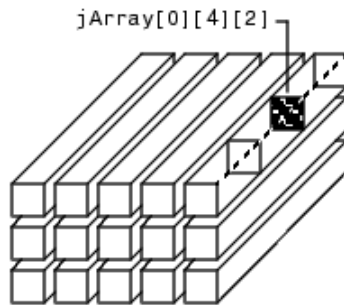
—jArray[0]   — jArray(1)
—jArray[1]   — jArray(2)
—jArray[2]   — jArray(3)

Simple Array   One-dimensional Array

jArray[0][3]   jArray(1,4)

Array of Arrays   Two-Dimensional Array

jArray[0][4][2]   jArray(1,5,3)
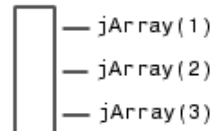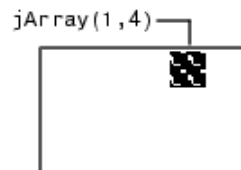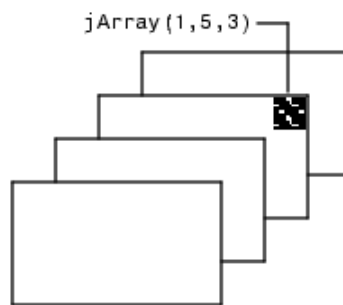
Array of Arrays of Arrays   Three-Dimensional Array

Array Indexing

Java array indexing is different than MATLAB array indexing. Java array indices are zero-based, MATLAB array indices are one-based. In Java programming, you access the elements of array y of length N using y[0]

through `y[N-1]`. When working with this array in MATLAB, you access these same elements using the MATLAB software indexing style of `y(1)` through `y(N)`. Thus, if you have a Java array of 10 elements, the seventh element is obtained using `y(7)`, and not `y[6]` as you use when writing a Java language program.

The Shape of the Java Array

A Java array can be different from a MATLAB array in its overall *shape*. A two-dimensional MATLAB array maintains a rectangular shape, as each row is of equal length and each column of equal height. The Java counterpart of this, an array of arrays, does not necessarily hold to this rectangular form. Each individual lower level array may have a different length.

Such an array structure is pictured below. This is an array of three underlying arrays of different lengths. The term *ragged* is commonly used to describe this arrangement of array elements as the array ends do not match up evenly. When a Java method returns an array with this type of structure, it is stored in a cell array by MATLAB.
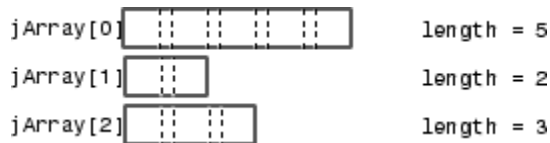
Interpreting the Size of a Java Array

When the MATLAB `size` function is applied to a simple Java array, the number of rows returned is the length of the Java array and the number of columns is always 1.

Determining the size of a Java array of arrays is not so simple. The potentially ragged shape of an array returned from a Java method makes it impossible to size the array in the same way as for a rectangular matrix. In a ragged Java array, there is no one value that represents the size of the lower level arrays.

When the `size` function is applied to a Java array of arrays, the resulting value describes the top level of the specified array. For the Java array:
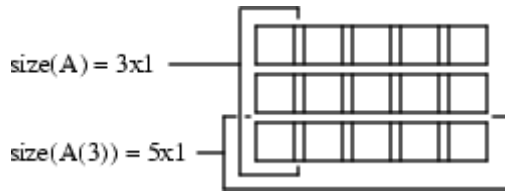
`size(A)` returns the dimensions of the highest array level of A. The highest level of the array has a size of 3-by-1.

```
size(A)
ans =
     3     1
```

To find the size of a lower level array, say the five-element array in row 3, refer to the row explicitly.

```
size(A(3))
ans =
     5     1
```

You can specify a dimension in the `size` command using the following syntax. However, you will probably find this useful only for sizing the first dimension, `dim=1`, as this will be the only non-unary dimension.

```
m = size(X,dim)

size(A, 1)
ans =
     3
```

Interpreting the Number of Dimensions of a Java Arrays

For Java arrays, whether they are simple one-level arrays or multilevel, the MATLAB `ndims` function always returns a value of 2 to indicate the number of dimensions in the array. This is a measure of the number of dimensions in the top-level array, which always equals 2.

Creating an Array of Objects in MATLAB Software

To call a Java method that has one or more arguments defined as an array of Java objects, you must, under most circumstances, pass your objects in a Java array. You can construct an array of objects in a call to a Java method or constructor. Or you can create the array within MATLAB.

The MATLAB `javaArray` function lets you create a Java array structure that can be handled in MATLAB as a single multidimensional array. You specify the number and size of the array dimensions along with the class of objects you intend to store in it. Using the one-dimensional Java array as its primary building block, MATLAB then builds an array structure that satisfies the dimensions requested in the `javaArray` command.

Using the javaArray Function

To create a Java object array, use the MATLAB `javaArray` function, which has the following syntax:

```
A = javaArray('element_class', m, n, p, ...)
```

The first argument is the `'element_class'` string, which names the class of the elements in the array. You must specify the fully qualified name (package and class name). The remaining arguments (`m, n, p, ...`) are the number of elements in each dimension of the array.

An array that you create with `javaArray` is equivalent to the array that you create with the Java code.

```
A = new element_class[m][n][p]...;
```

The following command builds a Java array of four lower level arrays, each capable of holding five objects of the `java.lang.Double` class. (You are more likely to use primitive types of `double` than instances of the `java.lang.Double` class, but in this context, it affords us a simple example.)

```
dblArray = javaArray('java.lang.Double', 4, 5);
```

The `javaArray` function does not deposit any values into the array elements that it creates. You must do this separately. The following MATLAB code stores objects of the `java.lang.Double` type in the Java array `dblArray` that was just created.

```
for m = 1:4
    for n = 1:5
    dblArray(m,n) = java.lang.Double((m*10) + n);
    end
end

dblArray
dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

Another Way to Create a Java Array

You can also create an array of Java objects using syntax that is more typical to MATLAB. For example, the following syntax creates a 4-by-5 MATLAB array of type double and assigns zero to each element of the array.

```
matlabArray(4,5) = 0;
```

You use similar syntax to create a Java array in MATLAB, except that you must specify the Java class name. The value being assigned, 0 in this example, is stored in the final element of the array, javaArray(4,5). All other elements of the array receive the empty matrix.

```
javaArray(4,5) = java.lang.Double(0)
javaArray =
java.lang.Double[][]:
    []      []      []      []      []
    []      []      []      []      []
    []      []      []      []      []
    []      []      []      []      [0]
```

Note You cannot change the dimensions of an existing Java array as you can with a MATLAB array. The same restriction exists when working with Java arrays in the Java language. See the example below.

This example first creates a scalar MATLAB array, and then successfully modifies it to be two-dimensional.

```
matlabArray = 0;
matlabArray(4,5) = 0

matlabArray =

     0    0    0    0    0
     0    0    0    0    0
     0    0    0    0    0
     0    0    0    0    0
```

When you try this with a Java array, you get an error message. Similarly, you cannot create an array of Java arrays from a Java array, and so forth.

```
javaArray = java.lang.Double(0);
javaArray(4,5) = java.lang.Double(0);
??? Index exceeds Java array dimensions.
```

Accessing Elements of a Java Array

You can access elements of a Java object array by using the MATLAB array indexing syntax, A(row,col). For example, to access the element of array dblArray located at row 3, column 4, use:

```
row3_col4 = dblArray(3,4)
row3_col4 =
34.0
```

In a Java language program, this is dblArray[2][3].

You can also use MATLAB array indexing syntax to access an element in an object's data field. Suppose that myMenuObj is an instance of a window menu class. This user-supplied class has a data field, menuItemArray, which is a Java array of java.awt.menuItem. To get element 3 of this array, use the following command.

```
currentItem = myMenuObj.menuItemArray(3)
```

Using Single Subscript Indexing to Access Arrays

Elements of a MATLAB matrix are most commonly referenced using both row and column subscripts. For example, you use x(3,4) to reference the array element at the intersection of row 3 and column 4. Sometimes it is more advantageous to use just a single subscript. MATLAB provides this capability (see the section on "Linear Indexing" in MATLAB Programming).

Indexing into a MATLAB matrix using a single subscript references one element of the matrix. Using the MATLAB matrix shown here, matlabArray (3) returns a single element of the matrix.

```
matlabArray = [11 12 13 14 15; 21 22 23 24 25; ...
               31 32 33 34 35; 41 42 43 44 45]
matlabArray =
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35
    41    42    43    44    45

matlabArray(3)
ans =
    31
```

Indexing this way into a Java array of arrays references an entire subarray of the overall structure. Using the dblArray Java array, that looks the same as matlabArray shown above, dblArray(3) returns the 5-by-1 array that makes up the entire third row.

```
row3 = dblArray(3)
row3 =
java.lang.Double[]:
    [31]
    [32]
    [33]
    [34]
    [35]
```

This is a useful feature of MATLAB because it allows you to specify an entire array from a larger array structure, and then manipulate it as an object.

Using the Colon Operator

Use of the MATLAB `colon` operator (`:`) is supported in subscripting Java array references. This operator works just the same as when referencing the contents of a MATLAB array. Using the Java array of `java.lang.Double` objects shown here, the statement `dblArray(2,2:4)` refers to a portion of the lower level array, `dblArray(2)`. A new array, `row2Array`, is created from the elements in columns 2 through 4.

```
dblArray
dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]

row2Array = dblArray(2,2:4)
row2Array =
java.lang.Double[]:
    [22]
    [23]
    [24]
```

You also can use the colon operator in single-subscript indexing, as covered in "Using Single Subscript Indexing to Access Arrays" on page 7-42. By making your subscript a colon rather than a number, you can convert an array of arrays into one linear array. The following example converts the 4-by-5 array `dblArray` into a 20-by-1 linear array.

```
linearArray = dblArray(:)
linearArray =
java.lang.Double[]:
    [11]
    [12]
    [13]
    [14]
    [15]
    [21]
    [22]
     .
     .
     .
```

This works the same way on an N-dimensional Java array structure. Using the colon operator as a single subscripted index into the array produces a linear array composed of all of the elements of the original array.

Note Java and MATLAB arrays are stored differently in memory. This is reflected in the order they are given in a linear array. Java array elements are stored in an order that matches the *rows* of the matrix, (11, 12, 13, ... in the array shown above). MATLAB array elements are stored in an order that matches the *columns*, (11, 21, 31, ...).

Using END in a Subscript

You can use the end keyword in the first subscript of an access statement. The first subscript references the top-level array in a multilevel Java array structure.

Note Using end on lower level arrays is not valid due to the potentially ragged nature of these arrays (see "The Shape of the Java Array" on page 7-37). In this case, there is no consistent end value to be derived.

The following example displays data from the third to the last row of Java array dblArray.

```
last2rows = dblArray(3:end, :)
last2rows =
java.lang.Double[][]:
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

Assigning to a Java Array

You assign values to objects in a Java array in essentially the same way as you do in a MATLAB array. Although Java and MATLAB arrays are structured quite differently, you use the same command syntax to specify which elements you want to assign to. See "Introduction" on page 7-34 for more information on Java and MATLAB array differences.

The following example deposits the value 300 in the dblArray element at row 3, column 2. In a Java language program, this is dblArray[2][1].

```
dblArray(3,2) = java.lang.Double(300)
dblArray =
java.lang.Double[][]:
    [11]    [ 12]    [13]    [14]    [15]
    [21]    [ 22]    [23]    [24]    [25]
    [31]    [300]    [33]    [34]    [35]
    [41]    [ 42]    [43]    [44]    [45]
```

You use the same syntax to assign to an element in an object's data field. Continuing with the myMenuObj example shown in "Accessing Elements of a Java Array" on page 7-41, you assign to the third menu item in menuItemArray as follows.

```
myMenuObj.menuItemArray(3) = java.lang.String('Save As...');
```

Using Single Subscript Indexing for Array Assignment

You can use a single-array subscript to index into a Java array structure that has more than one dimension. Refer to "Using Single Subscript Indexing to Access Arrays" on page 7-42 for a description of this feature as used with Java arrays.

You can use single-subscript indexing to assign values to an array as well. The example below assigns a one-dimensional Java array, onedimArray, to a row of a two-dimensional Java array, dblArray. Start out by creating the one-dimensional array.

```
onedimArray = javaArray('java.lang.Double', 5);
for k = 1:5
    onedimArray(k) = java.lang.Double(100 * k);
    end
```

Since dblArray(3) refers to the 5-by-1 array displayed in the third row of dblArray, you can assign the entire, similarly dimensioned, 5-by-1 onedimArray to it.

```
dblArray(3) = onedimArray
dblArray =
java.lang.Double[][]:
    [ 11]    [ 12]    [ 13]    [ 14]    [ 15]
    [ 21]    [ 22]    [ 23]    [ 24]    [ 25]
    [100]    [200]    [300]    [400]    [500]
    [ 41]    [ 42]    [ 43]    [ 44]    [ 45]
```

Assigning to a Linear Array

You can assign a value to *every* element of a multidimensional Java array by treating the array structure as if it were a single linear array. This entails replacing the single, numerical subscript with the MATLAB colon operator. If you start with the dblArray array, you can initialize the contents of every object in the two-dimensional array with the following statement.

```
dblArray(:) = java.lang.Double(0)
dblArray =
java.lang.Double[][]:
    [0]    [0]    [0]    [0]    [0]
    [0]    [0]    [0]    [0]    [0]
    [0]    [0]    [0]    [0]    [0]
    [0]    [0]    [0]    [0]    [0]
```

You can use the MATLAB colon operator as you would when working with MATLAB arrays. The statements below assign given values to each of the four rows in the Java array, dblArray. Remember that each row actually represents a separate Java array in itself.

```
dblArray(1,:) = java.lang.Double(125);
dblArray(2,:) = java.lang.Double(250);
dblArray(3,:) = java.lang.Double(375);
dblArray(4,:) = java.lang.Double(500)
dblArray =
java.lang.Double[][]:
    [125]    [125]    [125]    [125]    [125]
    [250]    [250]    [250]    [250]    [250]
    [375]    [375]    [375]    [375]    [375]
    [500]    [500]    [500]    [500]    [500]
```

Assigning the Empty Matrix

When working with MATLAB arrays, you can assign the empty matrix, (i.e., the 0-by-0 array denoted by [ ]) to an element of the array. For Java arrays, you can also assign [ ] to array elements. This stores the NULL value, rather than a 0-by-0 array, in the Java array element.

Subscripted Deletion

When you assign the empty matrix value to an entire row or column of a MATLAB array, you find that MATLAB actually removes the affected row or column from the array. In the example below, the empty matrix is assigned to all elements of the fourth column in the MATLAB matrix, matlabArray. Thus, the fourth column is completely eliminated from the matrix. This changes its dimensions from 4-by-5 to 4-by-4.

```
matlabArray = [11 12 13 14 15; 21 22 23 24 25; ...
               31 32 33 34 35; 41 42 43 44 45]
matlabArray =
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35
    41    42    43    44    45

matlabArray(:,4) = []
matlabArray =
    11    12    13    15
    21    22    23    25
    31    32    33    35
    41    42    43    45
```

You can assign the empty matrix to a Java array, but the effect is different. The next example shows that, when the same operation is performed on a Java array, the structure is not collapsed; it maintains its 4-by-5 dimensions.

```
dblArray(:,4) = []
dblArray =
java.lang.Double[][]:
    [125]    [125]    [125]    []    [125]
    [250]    [250]    [250]    []    [250]
    [375]    [375]    [375]    []    [375]
    [500]    [500]    [500]    []    [500]
```

The dblArray data structure is actually an array of five-element arrays of java.lang.Double objects. The empty array assignment placed the NULL value in the fourth element of each of the lower level arrays.

Concatenating Java Arrays

You can concatenate arrays of Java objects in the same way as arrays of other types. Java objects, however, can only be catenated along the first or second axis. To understand how scalar Java objects are concatenated in MATLAB software, see "Concatenating Java Objects" on page 7-18.

Use either the cat function or the square bracket ([ ]) operators. This example horizontally concatenates two Java arrays: d1 and d2.

```
% Construct a 2-by-3 array of java.lang.Double.
d1 = javaArray('java.lang.Double',2,3);
for m = 1:3      for n = 1:3
d1(m,n) = java.lang.Double(n*2 + m-1);
end;            end;

d1
d1 =
java.lang.Double[][]:
    [2]    [4]    [6]
    [3]    [5]    [7]
    [4]    [6]    [8]

% Construct a 2-by-2 array of java.lang.Double.
d2 = javaArray('java.lang.Double',2,2);
for m = 1:3      for n = 1:2
d2(m,n) = java.lang.Double((n+3)*2 + m-1);
end;            end;

d2
d2 =
java.lang.Double[][]:
    [ 8]    [10]
    [ 9]    [11]
    [10]    [12]

% Concatenate the two along the second dimension.
d3 = cat(2,d1,d2)
d3 =
java.lang.Double[][]:
    [2]    [4]    [6]    [ 8]    [10]
    [3]    [5]    [7]    [ 9]    [11]
    [4]    [6]    [8]    [10]    [12]
```

Creating a New Array Reference

Because Java arrays in MATLAB software are *references*, assigning an array variable to another variable results in a second reference to the array.

Consider the following example where two separate array variables reference a common array. The original array, origArray, is created and initialized.

The statement `newArrayRef = origArray` creates a copy of this array variable. Changes made to the array referred to by `newArrayRef` also show up in the original array.

```
origArray = javaArray('java.lang.Double', 3, 4);
for m = 1:3
   for n = 1:4
      origArray(m,n) = java.lang.Double((m * 10) + n);
   end
end

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]

%  ----- Make a copy of the array reference -----
newArrayRef = origArray;
newArrayRef(3,:) = java.lang.Double(0);

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [ 0]    [ 0]    [ 0]    [ 0]
```

Creating a Copy of a Java Array

You can create an entirely new array from an existing Java array by indexing into the array to describe a block of elements, (or subarray), and assigning this subarray to a variable. The assignment copies the values in the original array to the corresponding cells of the new array.

As with the example in section "Creating a New Array Reference" on page 7-49, an original array is created and initialized. But, this time, a copy is made of the array contents rather than copying the array reference. Changes made using the reference to the new array do not affect the original.

```
origArray = javaArray('java.lang.Double', 3, 4);
for m = 1:3
   for n = 1:4
      origArray(m,n) = java.lang.Double((m * 10) + n);
   end
end

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]

%  ----- Make a copy of the array contents -----
newArray = origArray(:,:);
newArray(3,:) = java.lang.Double(0);

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]
```

Passing Data to a Java Method

Introduction

When you make a call in the MATLAB software to Sun Java code, any MATLAB types you pass in the call are converted to 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 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 "Handling Data Returned from a Java Method" on page 7-63.

Conversion of MATLAB Argument Data

MATLAB data, passed as arguments to Java methods, are converted by MATLAB into 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.

Conversion of MATLAB Types to Java Types

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

Type conversion of arguments passed to Java code are discussed in the following three categories. MATLAB handles each category differently.

- "Passing Built-In Types" on page 7-54
- "Passing String Arguments" on page 7-55
- "Passing Java Objects" on page 7-56

Passing Built-In Types

The Java language has eight types that are intrinsic to the language and are not represented as Java objects. These are often referred to as *built-in*, or *elemental*, types and they include boolean, byte, short, long, int, double, float, and char. MATLAB software converts its own types to these Java built-in types according to the table, Conversion of MATLAB® Types to Java™ Types on page 7-53. Built-in types are in the first 10 rows of the table.

When a Java method you are calling expects one of these 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 type. MATLAB converts the 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 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 are not 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 7-34.

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 7-71. 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 Types" on page 7-54. The second, third, and fourth lines of

code shown here add items to a menu to be attached to the existing window frame. 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 following hash table example 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 type. Line 3, 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 creates 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 object arrays that you can pass to Java methods are Java arrays and MATLAB cell arrays. MATLAB automatically converts the cell array elements to `java.lang.Object` class objects. Note that in order for a cell array to be passed from MATLAB, the corresponding argument in the Java method signature must specify `java.lang.Object` or an array of `java.lang.Object`.

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

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

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

The MATLAB code shown below creates a 4-by-1 array containing four Java `Point` objects. When the array is passed to the `mapPoints` method, no conversion is necessary because the `javaArray` function created a Java array of `java.awt.Point` objects.

```
pointObj = javaArray('java.awt.Point',4);
pointObj(1) = java.awt.Point(25,143);
pointObj(2) = java.awt.Point(31,147);
pointObj(3) = java.awt.Point(49,151);
pointObj(4) = java.awt.Point(52,176);

testData.mapPoints(pointObj);
```

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 frameArray.

```
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,O,O,OxO,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
software 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, because 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 code, 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, 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 the Java language. The empty string (`' '`) in MATLAB translates into an empty (not `NULL`) `String` object in Java code.

Passing Data to Overloaded Methods

When you invoke an overloaded method on a Java object, the MATLAB software 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 7-58.

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 7-59.) 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 on page 7-53, 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 this section...
"Introduction" on page 7-63
"Conversion of Java Return Types" on page 7-63
"Built-In Types" on page 7-64
"Java Objects" on page 7-64
"Converting Objects to MATLAB Types" on page 7-65

Introduction

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

Conversion of Java Return Types

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.

Conversion of Java Types to MATLAB Types

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

Conversion of Java Types to MATLAB Types (Continued)

Java Return Type	If Scalar Return, Resulting MATLAB Type	If Array Return, Resulting MATLAB Type
double	double	double
char	char	char

Built-In Types

Java *built-in* types are described in "Passing Built-In Types" on page 7-54. This type includes boolean, byte, short, long, int, double, float, and char. When the value returned from a method call is one of these types, MATLAB software converts it according to the table Conversion of Java™ Types to MATLAB® Types on page 7-63.

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. An array of Java char converts to a MATLAB array of that type.

Java Objects

When a method call returns Java objects, the MATLAB software 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 7-63. 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 types listed in the right-hand column of the table. Other object types are not converted.

Converting Objects to MATLAB Types

With the exception of objects of class `Object`, MATLAB does not convert Java objects returned from method calls to a native MATLAB 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.

- "Converting to the MATLAB double Type" on page 7-65
- "Converting to the MATLAB char Type" on page 7-66
- "Converting to a MATLAB Structure" on page 7-66
- "Converting to a MATLAB Cell Array" on page 7-67

Converting to the MATLAB double Type

Using the `double` function in MATLAB, you can convert any Java object or array of objects to the MATLAB `double` 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), 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`. MATLAB uses `toDouble` to perform its conversion of Java objects to the MATLAB `double` 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 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 does not implement a `toDouble` method, then an attempt to convert the object using the `double` function results in a MATLAB error message.

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 Type

With the MATLAB `char` function, you can convert `java.lang.String` objects and arrays to MATLAB 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, MATLAB checks its class to see if it implements a method named `toChar`. If this is the case, MATLAB executes the `toChar` method of the class to perform the conversion. If you write your own class definitions, 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, an attempt to convert the object using the `char` function results in a MATLAB error message.

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 to do this.

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

```
cell(object);
```

The code in the following example creates a MATLAB cell array in which each cell holds an array of a different 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 =
F
    [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

- "Example — Reading a URL" on page 7-71
- "Example — Finding an Internet Protocol Address" on page 7-74
- "Example — Creating and Using a Phone Book" on page 7-76

Each example contains the following sections:

- Overview — Describes what the example does and how it uses the Sun 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.

Example — Reading a URL

In this section...

"Overview" on page 7-71

"Description of URLdemo" on page 7-71

"Running the Example" on page 7-72

Overview

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.

URLdemo constructs an object of the Sun 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 = 1:288          % Skip initial HTML formatting lines
   s = readLine(br);
end

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

Running the Example

When you run this example, you see output similar to the following. (Note that the line breaks were changed to fit the output in the documentation).

```
<p>This technical note provides an introduction to vectorization
```

techniques. In order to understand some of the possible techniques,
an introduction to MATLAB referencing is provided. Then several
vectorization examples are discussed.</p>
<p>This technical note examines how to identify situations where
vectorized techniques would yield a quicker or cleaner algorithm.
Vectorization is ofen a smooth process; however, in many
application-specific cases, it can be difficult to construct a
vectorized routine. Understanding the tools and

Example — Finding an Internet Protocol Address

In this section...
"Overview" on page 7-74
"Description of resolveip" on page 7-74
"Running the Example" on page 7-75

Overview

The resolveip function returns either the name or address of an IP (internet protocol) host. If you pass resolveip a host name, it returns the IP address. If you pass resolveip an IP address, it returns the host name. The function uses the Sun Java API class java.net.InetAddress, which enables you to find an IP address for a host name, or the host name 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 host name and IP address for the input argument. It displays either the host name 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 host name 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 host name and IP address.

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

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

3 Display the host name 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 host name 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 host name.

```
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
```

Example — Creating and Using a Phone Book

Overview

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 Sun 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 , it calls the java.lang.System static method getProperty to find the directory to use for the data dictionary. This is 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 path name, 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 `s`. 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.

```
case '1',
    name = input('Enter 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 entered, it passes it to `pb_remove`.

```
case '3',
    name=input('Enter name of 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 entered, it passes it to `pb_change`.

```
case '4',
    name=input('Enter name of 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;
```

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 is 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;
```

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 "Description of Function pb_lookup" on page 7-81.

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 of the entry, up to the line delimiter (^), into t 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
------------------------
```

COM Support for MATLAB Software

- "Introducing MATLAB COM Integration" on page 8-2
- "Getting Started with COM" on page 8-8
- "Supported Client/Server Configurations" on page 8-32

Introducing MATLAB COM Integration

In this section...
"What Is COM?" on page 8-2
"Concepts and Terminology" on page 8-2
"The MATLAB COM Client" on page 8-5
"The MATLAB COM Automation Server" on page 8-6
"Registering Controls and Servers" on page 8-6

What Is COM?

The Microsoft *Component Object Model* (*COM*) provides a framework for integrating reusable, binary software components into an application. Because components are implemented with compiled code, the source code can be written in any of the many programming languages that support COM. Upgrades to applications are simplified, as components can simply be swapped without the need to recompile the entire application. In addition, a component's location is transparent to the application, so components can 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 might require database access, mathematical analysis, and presentation-quality business graphs. Using COM, a developer can 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.

MATLAB software supports COM integration on the Microsoft Windows platform only.

Concepts and Terminology

While the ideas behind COM technology are straightforward, the terminology is not. The meaning of COM terms has changed over time and few concise definitions exist. Here are some terms that you should be familiar with before

reading this chapter. These are not comprehensive definitions. For a complete description of COM, you'll need to consult outside resources.

- "COM Objects, Clients, and Servers" on page 8-3

- "Interfaces" on page 8-3

- "COM Server Types" on page 8-4

- "Programmatic Identifiers" on page 8-4

- "In-Process and Out-of-Process Servers" on page 8-4

COM Objects, Clients, and Servers

A COM *object* is a software component that conforms to the Component Object Model. COM enforces encapsulation of the object, preventing direct access of its data and implementation. COM objects expose "Interfaces" on page 8-3, which consist of properties, methods and events.

A COM *client* is a program that makes use of COM objects. COM objects that expose functionality for use are called COM *servers*. COM servers can be in-process or out-of-process. An example of an out-of-process server is Microsoft® Excel® spreadsheet program. These configurations are described in "In-Process and Out-of-Process Servers" on page 8-4.

A Microsoft *ActiveX® control* is a type of in-process COM server that requires a control container. ActiveX controls typically have a user interface. An example is the Microsoft Calendar control. A control container is an application capable of hosting ActiveX controls. A MATLAB figure window or a Simulink model are examples of control containers.

MATLAB can be used as either a COM client or COM server.

Interfaces

The functionality of a component is defined by one or more interfaces. To use a COM component, you must learn about its interfaces, and the methods, properties, and events implemented by the component. The component vendor provides this information.

There are two standard COM interfaces:

- IUnknown — An interface required by all COM components. All other COM interfaces are derived from IUnknown.

- IDispatch — An interface that exposes objects, methods and properties to applications that support Automation.

COM Server Types

There are three types of COM servers:

- Automation — A server that supports the OLE Automation standard. Automation servers are based on the IDispatch interface. Automation servers can be accessed by clients of all types, including scripting clients.

- Custom — A server that implements an interface directly derived from IUnknown. Custom servers are preferred when faster client access is critical.

- Dual — A server that implements a combination of Automation and Custom interfaces.

Programmatic Identifiers

To create an instance of a COM object, you use its programmatic identifier, or *ProgID*. The ProgID is a unique string defined by the component vendor to identify the COM object. You obtain a ProgID from the vendor's documentation.

The MATLAB ProgIDs are

- Matlab.Application — Starts a command window Automation server with the version of MATLAB that was most recently used as an Automation server (might not be the latest installed version of MATLAB).

- Matlab.Autoserver — Starts a command window Automation server using the most recent version of MATLAB.

- Matlab.Desktop.Application — Starts the full desktop MATLAB as an Automation server using the most recent version of MATLAB.

In-Process and Out-of-Process Servers

You can configure a server three ways. MATLAB supports all of these configurations.

- "In-Process Server" on page 8-5
- "Local Out-of-Process Server" on page 8-5
- "Remote Out-of Process Server" on page 8-5

In-Process Server. An in-process server is a component implemented as a dynamic link library (DLL) or ActiveX control that runs in the same process as the client application, sharing the same address space. Communication between client and server is relatively fast and simple.

Local Out-of-Process Server. A local out-of-process server is a component implemented as an executable (EXE) file that runs in a separate process from the client application. The client and server processes are on the same computer system. This configuration is somewhat slower due to the overhead required when transferring data across process boundaries.

Remote Out-of Process Server. This is a type of out-of-process server; however, the client and server processes are on different systems and communicate over a network. Network communications, in addition to the overhead required for data transfer, can make this configuration slower than the local out-of-process configuration. This configuration runs only on systems that support the *Distributed Component Object Model* (*DCOM*).

The MATLAB COM Client

Using MATLAB as a COM client provides two techniques for developing programs in MATLAB:

- You can include COM components in your MATLAB application (for example, a spreadsheet).
- You can access existing applications that expose objects via Automation.

In a typical scenario, MATLAB creates ActiveX controls in figure windows, which are manipulated by MATLAB through the controls' properties, methods, and events. This is useful because there exists a wide variety of graphical user interface components implemented as ActiveX controls. For example, the Microsoft Internet Explorer® program exposes objects that you can include in a figure to display an HTML file. There also are treeviews, spreadsheets, and calendars available from a variety of sources.

MATLAB COM clients can access applications that support Automation, such as the Excel® spreadsheet program. In this case, MATLAB creates an Automation server in which to run the application and returns a handle to the primary interface for the object created.

Information about creating and using COM controls and server objects in MATLAB can be found in "Creating COM Objects" on page 9-2.

The MATLAB COM Automation Server

Automation provides an infrastructure whereby applications called automation controllers can access and manipulate (i.e. set properties of or call methods on) shared automation objects that are exported by other applications, called Automation servers. Any Windows program that can be configured as an Automation controller can control MATLAB.

For example, using Microsoft Visual Basic programming language, you can run a MATLAB demo in a Microsoft® PowerPoint® presentation. In this case, PowerPoint® is the controller and MATLAB is the server.

Information for creating and connecting to a MATLAB Automation server running MATLAB can be found in Chapter 10, "MATLAB COM Automation Server Support".

Registering Controls and Servers

Before using COM objects, you must register their controls and servers. Most are registered by default. However, if you get a new .ocx, .dll, or other object file for the control or server, you must register the file manually in the Windows registry.

Use the DOS regsvr32 command to register your file. From the DOS prompt, use the cd function to go to the directory where the object file is located. If your object file is an .ocx file, type:

```
regsvr32 filename.ocx
```

For example, to register the MATLAB control mwsamp2.ocx, type:

```
cd matlabroot\toolbox\matlab\winfun\win32
regsvr32 mwsamp2.ocx
```

If you encounter problems with this procedure, please consult a Windows manual or contact your local system administrator.

Verifying the Registration

Here are several ways to verify that a control or server is registered. These examples use the MATLAB `mwsamp` control. Refer to your Microsoft product documentation for information about using Microsoft Visual Studio or the Microsoft Registry Editor programs.

- Go to the Visual Studio .NET 2003 Tools menu and execute the ActiveX control test container. Click **Edit**, insert a new control, and select `MwSamp Control`. If you are able to insert the control without any problems, the control is successfully registered. Note that this method only works on controls.

- Open the Registry Editor by typing `regedit` at the DOS prompt. Search for your control or server object by selecting **Find** from the **Edit** menu. It will likely be in the following structure:

      HKEY_CLASSES_ROOT/progid

- Open OLEViewer from the Visual Studio .NET 2003 Tools menu. Look in the following structure for your `Control` object:

      Object Classes : Grouped by Component Category : Control :
      Your_Control_Object_Name (i.e. Object Classes : Grouped by
      Component Category : Control : Mwsamp Control)

Getting Started with COM

In this section...
"Introduction" on page 8-8
"Basic COM Functions" on page 8-8
"Overview of MATLAB COM Client Examples" on page 8-10
"Example — Using Internet Explorer Program in a MATLAB Figure" on page 8-11
"Example — Grid ActiveX Control in a Figure" on page 8-16
"Example — Reading Excel Spreadsheet Data" on page 8-24

Introduction

A COM client is a program that manipulates COM objects. These objects can run in the MATLAB application or can be part of another application that exposes its objects as a programmatic interface to the application.

This section provides examples that show how to use MATLAB as a COM client.

Note You can also access MATLAB as an Automation server from other applications, such as those written in the Microsoft Visual Basic programming language. For information on this technique, see Chapter 10, "MATLAB COM Automation Server Support".

Basic COM Functions

To start using COM objects, you need to create the object and get information about it. This section covers the following topics:

- "Creating an Instance of a COM Object" on page 8-9
- "Getting Information About a Particular COM Control" on page 8-9
- "Getting an Object's ProgID" on page 8-10

• "Registering a Custom Control" on page 8-10

Creating an Instance of a COM Object

Two MATLAB functions enable you to create COM objects:

• `actxcontrol` — Creates an instance of a control in a MATLAB figure.

• `actxserver` — Creates and manipulates objects from MATLAB that are exposed in an application that supports Automation.

Each function returns a *handle* to the object's main interface, which you use to access the object's methods, properties, and events, and any other interfaces it provides.

Getting Information About a Particular COM Control

In general, you can determine what you can do with an object using the `methods`, `get`, and `events` functions.

Information about Methods. To list the methods supported by the object *handle*, type:

    handle.methods

Information about Properties. To list the properties of the object *handle*, type:

    get(*handle*)

To see the value of the property *PropertyName*, type:

    get(*handle*,'*PropertyName*')

Use `set` to change a property value.

Information about Events. To list the events supported by the object *handle*, type:

    handle.events

For more information on calling syntax, see "Getting Interfaces to the Object" on page 9-69 and "Invoking Methods on an Object" on page 9-44. For more information on events, see "Using Events" on page 9-52.

Getting an Object's ProgID

To get the programmatic identifier (ProgID) of a COM control that is already registered on your computer, use the actxcontrollist command. You can also use the **ActiveX Control Selector**, displayed with the command actxcontrolselect. This interface lets you see instances of the controls installed on your computer.

For more information on using these commands, see "Creating an ActiveX Control" on page 9-3.

Registering a Custom Control

If your MATLAB program uses a custom control (e.g., one that you have created especially for your application), you must register it with the Microsoft Windows operating system before you can use it. You can do this from your MATLAB program by issuing an operating system command:

```
!regsvr32 /s filename.ocx
```

where *filename* is the name of the file containing the control. Using this command in your program enables you to provide custom-made controls that you make available to other users by registering the control on their computer when they run your MATLAB program. You might also want to supply versions of a Microsoft ActiveX control to ensure that all users have the same version.

For more information about registration, see "Registering Controls and Servers" on page 8-6.

Overview of MATLAB COM Client Examples

The following examples illustrate various techniques for using MATLAB software as a COM client. Some of the examples use ActiveX controls, which is a specific type of COM object. For a description, see "COM Objects, Clients, and Servers" on page 8-3.

- "Example — Using Internet Explorer Program in a MATLAB Figure" on page 8-11 — This example uses the ActiveX control exposed by Internet Explorer web browser to add an HTML viewer to a MATLAB Figure, which also contains an axes object for plotting. As the user clicks various graphics objects that are displayed in the figure (including the figure itself), the documentation of the object's properties is displayed in the viewer.

- "Example — Grid ActiveX Control in a Figure" on page 8-16 — This example puts a spreadsheet-like grid control in a figure and uses the control's mouse-down event to trigger the acquisition of data from the grid and plot the data in the axes.

- "Example — Reading Excel Spreadsheet Data" on page 8-24 — This MATLAB GUI reads data programmatically from an Excel spreadsheet. By running an Automation server, the MATLAB software can access the objects exposed by the spreadsheet program, which provides a variety of interfaces to the application.

Example — Using Internet Explorer Program in a MATLAB Figure

This example uses the ActiveX control `Shell.Explorer`, which is exposed by theMicrosoft Internet Explorer application, to include an HTML viewer in a MATLAB figure. The figure's window button down function is then used to select a graphics object when the user clicks the graph and load the object's property documentation into the HTML viewer.

Techniques Demonstrated

- Using Internet Explorer from an ActiveX client program.

- Defining a window button down function that displays HTML property documentation for whatever object the user clicks.

- Defining a resize function for the figure that also resizes the ActiveX object container.

Using the Figure to Access Properties

This example creates a larger than normal figure window that contains an axes object and an HTML viewer on the lower part of the figure window. By

default, the viewer displays the URL `http://www.mathworks.com`. When you issue a plotting command, such as:

```
surfc(peaks(20))
```

the graph displays in the axes.

Click anywhere in the graph to see the property documentation for the selected object.
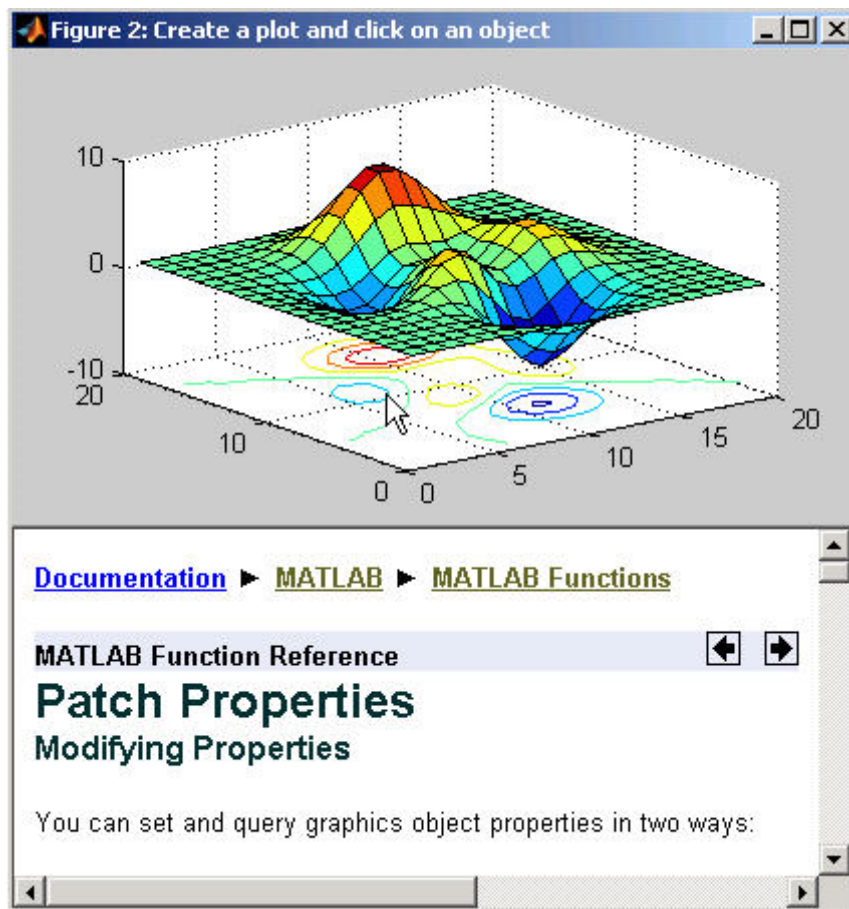
Complete Code Listing

You can open the M-file that implements this example in the MATLAB Editor or you can run this example with the following links:

- Open M-file in editor

- Run this example

Creating the Figure

This example defines the figure size based on the default figure size and adds space for the ActiveX control. Here is the code to define the figure:

```
dfpos = get(0,'DefaultFigurePosition');
hfig = figure('Position',dfpos([1 2 3 4]).*[.8 .2 1 1.65],...
    'Menu','none','Name','Create a plot and click on an object',...
    'ResizeFcn',@reSize,...
    'WindowButtonDownFcn',@wbdf,...
    'Renderer','Opengl',...
    'DeleteFcn',@figDelete);
```

Note that the figure also defines a resize function and a window button down function by assigning function handles to the `ResizeFcn` and `WindowButtonDownFcn` properties. The callback functions `reSize` and `wbdf` are defined as nested functions in the same M-file.

The figure's `delete` function (called when the figure is closed) provides a mechanism to delete the control.

Calculating the ActiveX Object Container Size

The `actxcontrol` function creates the ActiveX control inside the specified figure and returns the control's handle. You need to supply the following information:

- Control's programmatic identifier (use `actxcontrollist` to find it)

- Location and size of the control container in the figure (pixels) [left bottom width height]

- Handle of the figure that contains the control:

```
conSize = calcSize;                                    % Calculate the container size
hExp = actxcontrol('Shell.Explorer.2',conSize,hfig);  % Create the control
Navigate(hExp,'http://www.mathworks.com/');            % Specify content of html viewer
```

The nested function, `calcSize` calculates the size of the object container based on the current size of the figure. `calcSize` is also used by the figure `resize` function, which is described in the next section.

```
function conSize = calcSize
fp = get(hfig,'Position');              % Get current figure size
conSize = [0 0 1 .45].*fp([3 4 3 4]);   % Calculate container size
end % calcSize
```

Automatic Resize

In MATLAB, you can change the size of a figure and the axes automatically resize to fit the new size. This example implements similar resizing behavior for the ActiveX object container within the figure using the object's `move` method. This method enables you to change both size and location of the ActiveX object container (i.e., it is equivalent to setting the figure `Position` property).

When you resize the figure window, the MATLAB software automatically calls the function assigned to the figure's `ResizeFcn` property. This example implements the nested function `reSize` for the figure `reSize` function.

ResizeFcn at Figure Creation. The `resize` function first determines if the ActiveX object exists because the MATLAB software calls the figure `resize` function when the figure is first created. Since the ActiveX object has not been created at this point, the `resize` function simply returns.

When the Figure Is Resized. When you change the size of the figure, the `resize` function executes and does the following:

• Calls the `calcSize` function to calculate a new size for the control container based on the new figure size.

• Calls the control's `move` method to apply the new size to the control.

Figure ResizeFcn.

```
function reSize(src,evnt)
if ~exist('hExp','var')
   return
end
conSize = calcSize;
move(hExp,conSize);
end % reSize
```

Selecting Graphics Objects

This example uses the figure WindowButtonDownFcn property to define a callback function that handles mouse click events within the figure. When you click the left mouse button while the cursor is over the figure, the MATLAB software executes the WindowButtonDownFcn callback on the mouse down event.

The callback determines which object was clicked by querying the figure CurrentObject property, which contains the handle of the graphics object most recently clicked. Once you have the object's handle, you can determine its type and then load the appropriate HTML page into the Shell.Explorer control.

The nested function wbdf implements the callback. Once it determines the type of the selected object, it uses the control Navigate method to display the documentation for the object type.

Figure WindowButtonDownFcn.

```
function wbdf(src,evnt)
cobj = get(hfig,'CurrentObject');
if isempty(cobj)
   disp('Click somewhere else')
   return
end
pth = 'http://www.mathworks.com/access/helpdesk/help/techdoc/ref/';
typ = get(cobj,'Type');
switch typ
   case ('figure')
      Navigate(hExp,[pth,'figure_props.html']);
```

```
     case ('axes')
        Navigate(hExp,[pth,'axes_props.html']);
     case ('line')
        Navigate(hExp,[pth,'line_props.html']);
     case ('image')
        Navigate(hExp,[pth,'image_props.html']);
     case ('patch')
        Navigate(hExp,[pth,'patch_props.html']);
     case ('surface')
        Navigate(hExp,[pth,'surface_props.html']);
     case ('text')
        Navigate(hExp,[pth,'text_props.html']);
     case ('hggroup')
        Navigate(hExp,[pth,'hggroupproperties.html']);
     otherwise % Display property browser
        Navigate(hExp,[pth(1:end-4),'infotool/hgprop/doc_frame.html']);
  end
  end % wbdf
```

Closing the Figure

This example uses the figure `delete` function (`DeleteFcn` property) to delete the ActiveX object before closing the figure. The MATLAB software calls the figure `delete` function before deleting the figure, which enables the function to perform any clean up needed before closing the figure. The figure `delete` function calls the control's `delete` method.

```
function figDelete(src,evnt)
   hExp.delete;
end
```

Example — Grid ActiveX Control in a Figure

This example adds a Microsoft ActiveX spreadsheet control to a figure, which also contains an axes object for plotting the data displayed by the control. Clicking a column in the spreadsheet causes the data in that column to be plotted. Clicking down and dragging the mouse across multiple columns plots all columns touched.

Techniques Demonstrated

- Registering a control for use on your system.

- Writing a handler for one of the control's events and using the event to execute MATLAB plotting commands.

- Writing a `resize` function for the figure that manages the control's size as users resize the figure.

Using the Control

This example assumes that your data samples are organized in columns and that the first cell in each column is a title, which is used by the legend. See "Complete Code Listing" on page 8-18 for an example of how to load data into the control.

Once the data is loaded, click the column to plot the data. The following picture shows a graph of the results of `Test2` and `Test3` created by selecting column `B` and dragging and releasing on column `C`.
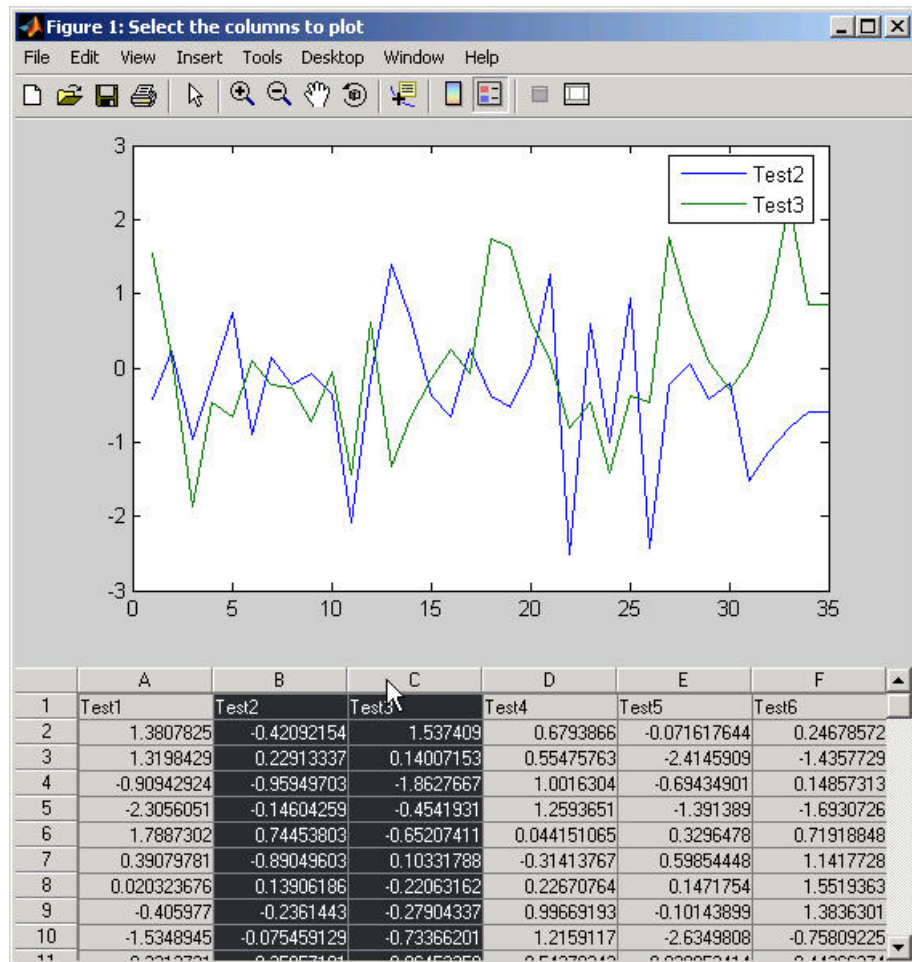
Complete Code Listing

You can open the M-file used to implement this example in the MATLAB Editor:

- Open M-file in editor.

Preparing to Use the Control

The ActiveX control used in this example is typical of those downloadable from the Internet. Once you have downloaded the files you need, register the control on your system using the DOS command `regsvr32`. In a command prompt, enter a command of the following form:

```
regsvr32 sgrid.ocx
```

From the MATLAB command line, type:

```
system 'regsvr32 sgrid.ocx'
```

See the section "Registering Controls and Servers" on page 8-6 for more information.

Finding the Control's ProgID. Once you have installed and registered the control, you can obtain its programmatic identifier using the **ActiveX Control Selector** dialog. To display this dialog box, use the `actxcontrolselect` command. Locate the control in the list and the selector displays the control and the ProgID.
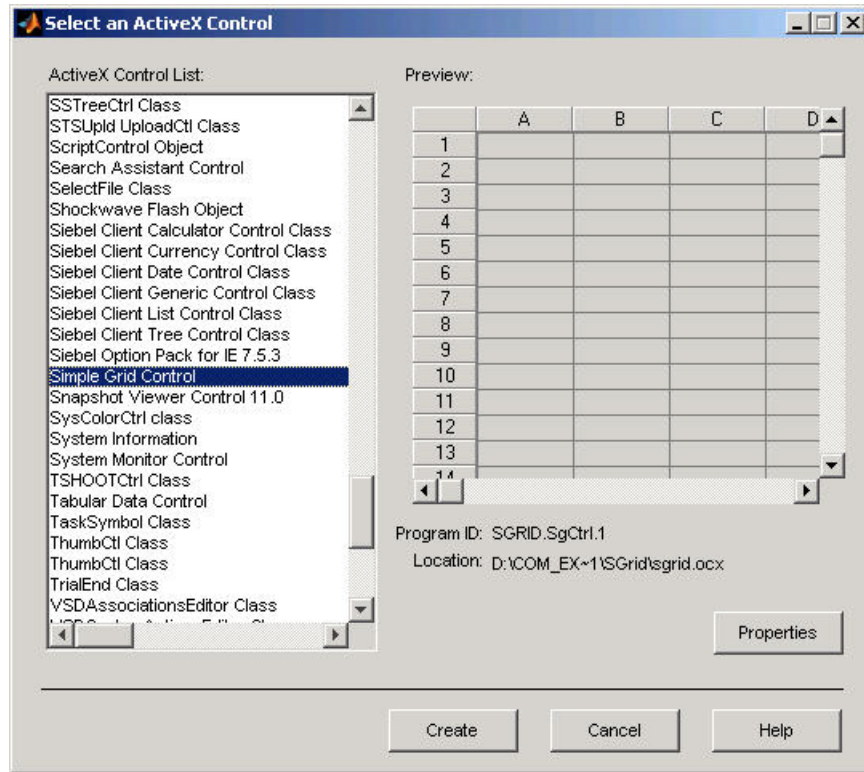
Creating a Figure to Contain the Control

This example creates a figure that contains an axes object and the grid control. The first step is to determine the size of the figure and then create the figure and axes. This example uses the default figure and axes size (obtained from the respective Position properties) to calculate a new size and location for each object.

```
dfpos = get(0,'DefaultFigurePosition');
dapos = get(0,'DefaultAxesPosition');
hfig = figure('Position',dfpos([1 2 3 4]).*[1 .8 1 1.25],...
    'Name','Select the columns to plot',...
    'Renderer','ZBuffer',...
    'ResizeFcn',{@reSize dfpos(3)});
hax = axes('Position',dapos([1 2 3 4]).*[1 4 1 .65]);
```

The above code moves the figure down from the top of the screen (multiply second element in position vector by .8) and increases the height of the figure (multiply fourth element in position vector by 1.25). Axes are created and sized in a similar way.

Creating an Instance of the Control

Use the actxcontrol function to create an instance of the control in a figure window. This function creates a container for the control and enables you to specify the size of this container, which usually defines the size of the control. See "Managing Figure Resize" on page 8-23 for a specific example.

Specifying the Size and Location. The control size and location in the figure is calculated by a nested function calcSize. This function is used to calculate both the initial size of the control container and the size resulting from resize of the figure. It gets the figure's current position (i.e., size and location) and scales the four-element vector so that the control container is

- Positioned at the lower-left corner of the figure.

- Equal to the figure in width.

- Has a height that is .35 times the figure's height.

The value returned is of the correct form to be passed to the actxcontrol function and the control's move method.

```
function conSize = calcSize
   fp = get(hfig,'Position');
   conSize = fp([3 4 3 4]).*[0 0 1 .35];
end % conSize
```

Creating the Control. Creating the control entails the following steps:

- Calculating the container size

- Instantiating the control in the figure

- Setting the number of rows and columns to match the size of the data array

- Specifying the width of the columns

```
conSize = calcSize;
```

```
hgrid = actxcontrol('SGRID.SgCtrl.1',conSize,hfig);
hgrid.NRows = size(dat,1);
hgrid.NColumns = size(dat,2);
colwth = 4350; hdwth = hgrid.HdrWidth;
SetColWidth(hgrid,0,sz(2)-1,colwth,1)
```

Using Mouse-Click Event to Plot Data

This example uses the control's `Click` event to implement interactive plotting. When a user clicks the control, the MATLAB software executes a function that plots the data in the column where the mouse click occurred. Users can also select multiple columns by clicking down and dragging the cursor over more than one column.

Registering the Event. You need to register events with MATLAB so that when the event occurs (a mouse click in this case), the MATLAB software responds by executing the event handler function. Register the event with the `registerevent` function:

```
hgrid.registerevent({'Click',@click_event});
```

Pass the event name (`Click`) and a function handle for the event handler function inside a cell array.

Defining the Event Handler. The event handler function `click_event` uses the control's `GetSelection` method to determine what columns and rows have been selected by the mouse click. This function plots the data in the selected columns as lines, one line per column.

It is possible to click down on a column and drag the mouse to select multiple columns before releasing the mouse. In this case, each column is plotted because the event is not fired until the mouse button is released (which reflects the way the author chose to implement the control). The `legend` function uses the column number stored in the variable `cols` to label the individual plotted lines. You must add one to `cols` because the control counts the columns starting from zero.

Note that you implement event handlers to accept a variable number of arguments (`varargin`).

```
function click_event(varargin)
```

```
[row1,col1,row2,col2] = hgrid.GetSelection(1,1,1,1,1);
ncols = (col2-col1)+1;
cols = [col1:col2];
   for n = 1:ncols
      hgrid.Col = cols(n);
      for ii = 1:sz(1)
         hgrid.Row = ii;
         plot_data(ii,n) = hgrid.Number;
      end
   end
hgrid.SetSelection(row1,col1,row2,col2);
plot(plot_data)
legend(labels(cols+1))
end % click_event
```

Managing Figure Resize

The size and location of a MATLAB axes object is defined in units that are normalized to the figure that contains it. Therefore, when you resize the figure, the axes automatically resize proportionally. When a figure contains objects that are not contained in axes, you are responsible for defining a function that manages the resizing process.

The figure ResizeFcn property references a function that executes whenever the figure is resized and also when the figure is first created. This example creates a resize function that manages resizing grid control by doing the following:

- Disables control updates while changes are being made to improve performance (use the hDisplay property).

- Calculates a new size for the control container based on the new figure size (calcSize function).

- Applies the new size to the control container using its move method.

- Scales the column widths of the grid proportional to the change in width of the figure (SetColWidth method).

- Refreshes the display of the control, showing the new size.

```
function reSize(src,evnt,dfp)
```

```
% Return if control does not exist (figure creation)
if ~exist('hgrid','var')
   return
end
% Resize container
hgrid.bDisplay = O;
conSize = calcSize;
move(hgrid,conSize);
% Resize columns
scl = conSize(3)/dfp;
ncolwth = scl*colwth;
nhdrwth = hdwth*(scl);
hgrid.HdrWidth = nhdrwth;
SetColWidth(hgrid,O,sz(2)-1,ncolwth,2)
hgrid.Refresh;
end % reSize
```

Closing the Figure

This example uses the figure delete function (`DeleteFcn` property) to delete the ActiveX object before closing the figure. The MATLAB software calls the figure delete function before deleting the figure, which enables the function to perform any clean up needed before closing the figure. The figure delete function calls the control's `delete` method.

```
function figDelete(src,evnt)
   hgrid.delete;
end
```

Example — Reading Excel Spreadsheet Data

This example creates a graphical interface to access the data in a Microsoft Excel file. To enable the communication between the MATLAB software and the spreadsheet program, this example creates an Microsoft ActiveX object in an Automation server running an Excel application. The MATLAB software then accesses the data in the spreadsheet through the interfaces provided by the Excel Automation server.

Techniques Demonstrated

This example shows how to use the following techniques:

- Use of an Automation server to access another application from the MATLAB software.

- Ways to manipulate Excel data into types used in the GUI and plotting.

- Implementing a GUI that enables plotting of selected columns of the Excel spreadsheet.

- Inserting a MATLAB figure into an Excel file.

Using the GUI

To use the GUI, select any items in the list box and click the **Create Plot** button. The sample data provided with this example contain three input and three associated response data sets, all of which are plotted versus the first column in the Excel file, which is the time data.

You can view the Excel data file by clicking the **Show Excel Data File** button, and you can save an image of the graph in a different Excel file by clicking **Save Graph** button. Note that the **Save Graph** option creates a temporary PNG file in your working directory.

The following picture shows the GUI with an input/response pair selected in the list box and plotted in the axes.
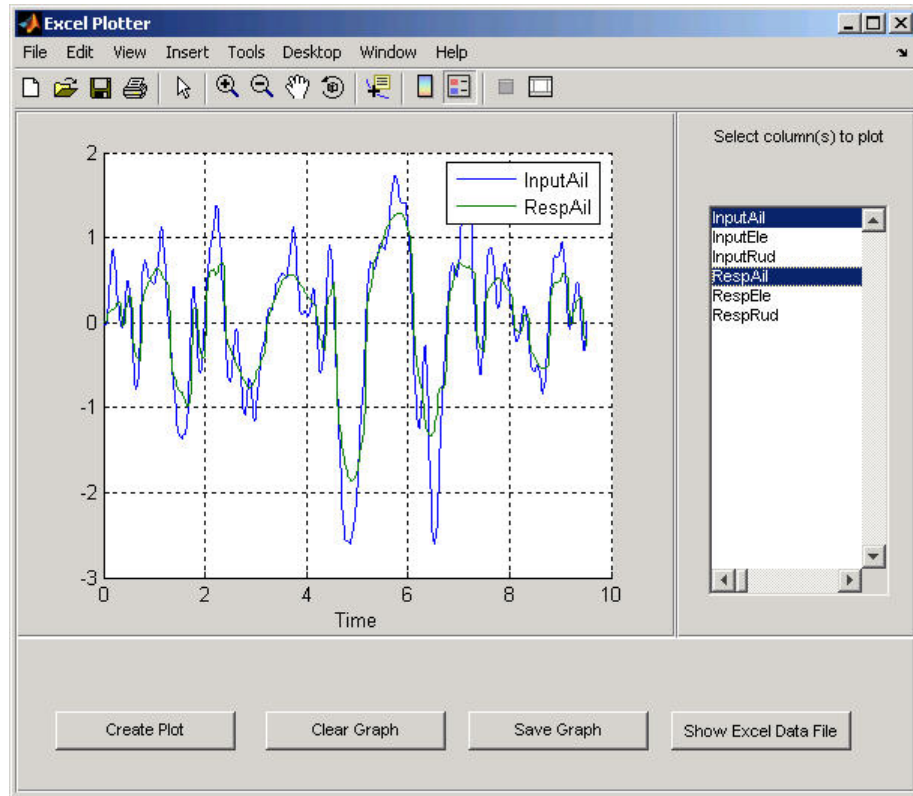
Complete Code Listing

You can open the M-file used to implement this example in the MATLAB Editor or run this example:

• Open M-file in editor.

• Run this example.

Excel Spreadsheet Format

This example assumes a particular organization of the Excel spreadsheet, as shown in the following picture.

The format of the Excel file is as follows:

- The first element in each column is a text string that identifies the data contain in the column. These strings are extracted and used to populate the list box.

- The first column (Time) is used for the *x*-axis of all plots of the remaining data.

- All rows in each column are read into the MATLAB software.

Excel Automation Server

The first step in accessing the spreadsheet data from the MATLAB software is to run the Excel application in an Automation server process using the actxserver function and the program ID, excel.application.

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

The ActiveX object that is returned provides access to a number of interfaces supported by the Excel program. Use the workbook interface to open the Excel file containing the data.

```
exlWkbk = exl.Workbooks;
exlFile = exlWkbk.Open([docroot '/techdoc/matlab_external/examples/input_resp_data.xls']);
```

Use the workbook's sheet interface to access the data from a range object, which stores a reference to a range of data from the specified sheet. This example accesses all the data in column A, first cell to column G, last cell:

```
exlSheet1 = exlFile.Sheets.Item('Sheet1');
robj = exlSheet1.Columns.End(4);       % Find the end of the column
numrows = robj.row;                    % And determine what row it is
dat_range = ['A1:G' num2str(numrows)]; % Read to the last row
rngObj = exlSheet1.Range(dat_range);
```

At this point, the entire data set from the Excel file's sheet1 is accessed via the range object interface. This object returns the data in a MATLAB cell array, which can contain both numeric and character data:

```
exlData = rngObj.Value;
```

Manipulating the Data in the MATLAB Workspace

Now that the data is in a cell array, you can use MATLAB functions to extract and reshape parts of the data into the forms needed to use in the GUI and pass to the plot function.

The following code performs two operations:

- Extracts numeric data from the cell array (indexing with curly braces), concatenates the individual doubles returned by the indexing operation (square brackets), and reshapes it into an array that arranges the data in columns.

- Extracts the string in the first cell in each column of an Excel sheet and stores them in a cell array, which is used to generate the items in the list box.

```
for ii = 1:size(exlData,2)
   matData(:,ii) = reshape([exlData{2:end,ii}],size(exlData(2:end,ii)));
```

```
    lBoxList{ii} = [exlData{1,ii}];
end
```

The Plotter GUI

This example uses a GUI that enables you to select from a list of input and response data from a list box. All data is plotted as a function of time (which is, therefore, not a choice in the list box) and you can continue to add more data to the graph. Each data plot added to the graph causes the legend to expand.

Additional implementation details include:

• A legend that updates as you add data to a graph

• A clear button that enables you to clear all graphs from the axes

• A save button that saves the graph as a PNG file and adds it to another Excel file

• A toggle button that shows or hides the Excel file being accessed

• The figure delete function (DeleteFcn property), which the MATLAB software calls when the figure is closed, is used to terminate the Automation server process.

Selecting and Plotting Data. When you click the **Create Plot** button, its callback function queries the list box to determine what items are selected and plots each data versus time. The legend is updated to display any new data while maintaining the legend for the existing data.

```
function plotButtonCallback(src,evnt)
iSelected = get(listBox,'Value');
grid(a,'on');hold all
for p = 1:length(iSelected)
   switch iSelected(p)
      case 1
         plot(a,tme,matData(:,2))
      case 2
         plot(a,tme,matData(:,3))
      case 3
         plot(a,tme,matData(:,4))
      case 4
```

```
                plot(a,tme,matData(:,5))
            case 5
                plot(a,tme,matData(:,6))
            case 6
                plot(a,tme,matData(:,7))
            otherwise
                disp('Select data to plot')
        end
    end
    [b,c,g,lbs] = legend([lbs lBoxList(iSelected+1)]);
    end % plotButtonCallback
```

Clearing the Axes. The plotter is designed to continually add graphs as the user selects data from the list box. The **Clear Graph** button clears and resets the axes and clears the variable used to store the labels of the plot data (used by legend).

```
%% Callback for clear button
function clearButtonCallback(src,evt)
    cla(a,'reset')
    lbs = '';
end % clearButtonCallback
```

Display or Hide Excel File. The MATLAB program has access to the properties of the Excel application running in the Automation server. By setting the Visible property to 1 or 0, this callback controls the visibility of the Excel file.

```
%% Display or hide Excel file
function dispButtonCallback(src,evt)
    exl.visible = get(src,'Value');
end % dispButtonCallback
```

Close Figure and Terminate Excel Automation Process. Since the Excel Automation server runs in a separate process from the MATLAB software, you must terminate this process explicitly. There is no reason to keep this process running after the GUI has been closed, so this example uses the figure's delete function to terminate the Excel process with the Quit method. Also, terminate the second Excel process used for saving the graph. See "Inserting MATLAB Graphs Into Excel Spreadsheets" on page 8-31 for information on this second process.

```
%% Terminate Excel processes
function deleteFig(src,evt)
   exlWkbk.Close
   exlWkbk2.Close
   exl.Quit
   exl2.Quit
end % deleteFig
```

Inserting MATLAB Graphs Into Excel Spreadsheets

You can save the graph created with this GUI in an Excel file. (This example uses a separate Excel Automation server process for this purpose.) The callback for the **Save Graph** push button creates the image and adds it to an Excel file:

- Both the axes and legend are copied to an invisible figure configured to print the graph as you see it on the screen (figure PaperPositionMode property is set to auto).

- The print command creates the PNG image.

- Use the Shapes interface to insert the image in the Excel workbook.

The server and interfaces are instanced during GUI initialization phase:

```
exl2 = actxserver('excel.application');
exlWkbk2 = exl2.Workbooks;
wb = invoke(exlWkbk2,'Add');
graphSheet = invoke(wb.Sheets,'Add');
Shapes = graphSheet.Shapes;
```

The following code implements the **Save Graph** button callback:

```
function saveButtonCallback(src,evt)
   tempfig = figure('Visible','off','PaperPositionMode','auto');
   tempfigfile = [tempname '.png'];
   ah = findobj(f,'type','axes');
   copyobj(ah,tempfig) % Copy both graph axes and legend axes
   print(tempfig,'-dpng',tempfigfile);
   Shapes.AddPicture(tempfigfile,0,1,50,18,300,235);
   exl2.visible = 1;
end
```

Supported Client/Server Configurations

Introduction

You can configure MATLAB software 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 another component controls MATLAB, MATLAB is the server.

MATLAB Client and In-Process Server

The following diagram shows how the MATLAB client interacts with an "In-Process Server" on page 8-5.
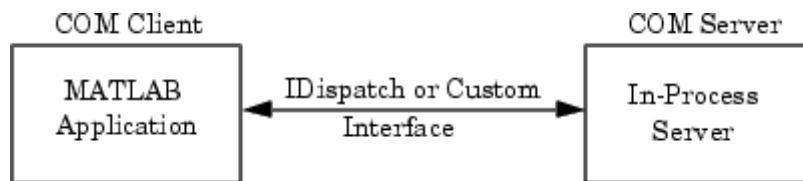
The server exposes its properties and methods through the IDispatch (Automation) interface or a Custom interface, depending on which interfaces the component implements. For information on accessing interfaces, see "Getting Interfaces to the Object" on page 9-69 .

Microsoft ActiveX Controls

An ActiveX control is an object with some type of graphical user interface (GUI). When the MATLAB software constructs an ActiveX control, it places the control's GUI in a MATLAB figure window. Click the various options available in the user interface (e.g., making a particular menu selection) to trigger *events* that get communicated from the control in the server to the client MATLAB application. The client decides what to do about each event and responds accordingly.

MATLAB comes with a sample ActiveX control called `mwsamp`. This control draws a circle on the screen and displays some text. You can use this control to try out MATLAB COM features. For more information, see "MATLAB Sample Control" on page 9-84.

DLL Servers

Any COM component that has been implemented as a dynamic link library (DLL) is also instantiated in an in-process server. That is, it is created in the same process as the MATLAB client application. When MATLAB uses a DLL server, it runs in a separate window rather than a MATLAB figure window.

MATLAB responds to events generated by a DLL server in the same way as events from an ActiveX control.

For More Information

To learn more about working with MATLAB as a client, see "Creating COM Objects" on page 9-2 and "Advanced Topics" on page 9-89.

MATLAB Client and Out-of-Process Server

In this configuration, a MATLAB client application interacts with a component that has been implemented as a "Local Out-of-Process Server" on page 8-5. Examples of out-of-process servers are Microsoft Excel and Microsoft Word programs.
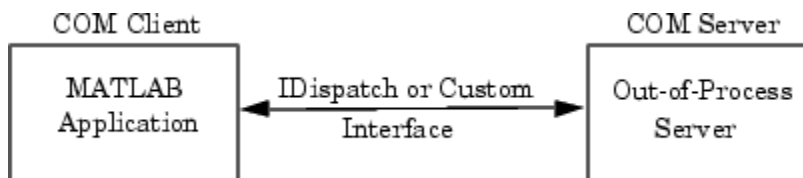
As with in-process servers, this server exposes its properties and methods through the IDispatch (Automation) interface or a Custom interface, depending on which interfaces the component implements. For information on accessing interfaces, see "Getting Interfaces to the Object" on page 9-69.

Since the client and server run in separate processes, you have the option of creating the server on any system on the same network as the client.

If the component provides a user interface, its window is separate from the client application.

MATLAB responds to events generated by an out-of-process server in the same way as events from an ActiveX control.

For More Information

To learn more about working with MATLAB as a client, see "Creating COM Objects" on page 9-2 and "Advanced Topics" on page 9-89.

COM Implementations Supported by MATLAB Software

MATLAB only supports COM implementations that are compatible with the Microsoft Active Template Library (ATL) API. In general, your COM object should be able to be contained in an ATL host window in order to work with MATLAB.

Client Application and MATLAB Automation Server

MATLAB operates as the Automation server in this configuration. It can be created and controlled by any Microsoft Windows program that can be an *Automation controller*. Some examples of Automation controllers are

Microsoft Excel, Microsoft® Access™, Microsoft Project, and many Microsoft Visual Basic and Microsoft Visual C++ programs.
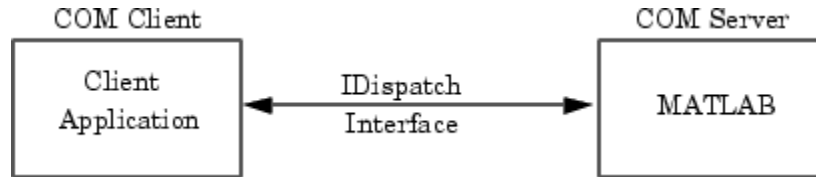
MATLAB Automation server capabilities include the ability to execute commands in the MATLAB workspace, and to get and put matrices directly from and into the workspace. You can start a MATLAB server to run in either a shared or dedicated mode. You also have the option of running it on a local or remote system.

To create the MATLAB server from an external application program, use the appropriate function from that language to instantiate the server. (For example, use the Visual Basic® `CreateObject` function.) For the programmatic identifier, specify `matlab.application`. To run MATLAB as a dedicated server, use the `matlab.application.single` programmatic identifier. See "Using MATLAB Software as a Shared or Dedicated Server" on page 10-3 for more information.

The function that creates the MATLAB server also returns a handle to the properties and methods available in the server through the IDispatch interface. See "MATLAB Automation Server Functions and Properties" on page 10-7 for descriptions of these methods.

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

For More Information

To learn more about working with Automation servers, see Chapter 10, "MATLAB COM Automation Server Support" and "Additional Automation Server Information" on page 10-13.

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 IEngine to communicate between the client application and the MATLAB engine running as a COM server.
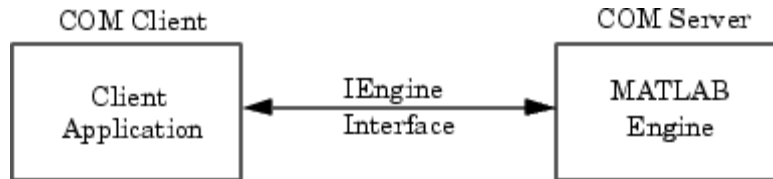
MATLAB provides a library of functions that let you to start and end the server process, and to send commands to be processed by MATLAB. See "MATLAB Engine" in the C and Fortran API Reference for more information.

For More Information

To learn more about the MATLAB engine and the functions provided in its C and Fortran libraries, see Chapter 6, "Calling MATLAB Software from C and Fortran Programs".

9

MATLAB COM Client Support

Creating COM Objects

In this section...
"Creating the Server Process — An Overview" on page 9-2
"Creating an ActiveX Control" on page 9-3
"Creating a COM Server" on page 9-9

Creating the Server Process — An Overview

MATLAB software provides two functions to create a COM object:

- actxcontrol — Creates a Microsoft ActiveX control in a MATLAB figure window.

- actxserver — Creates an in-process server for a dynamic link library (DLL) component or an out-of-process server for an executable (EXE) component.

The following diagram shows the basic steps in creating the server process. For more information on how the MATLAB software establishes interfaces to the resultant COM object, see "Getting Interfaces to the Object" on page 9-69.

```
┌─────────────────────────────┐                    ┌─────────────────────────────┐
│      Enter the command      │                    │      Enter the command      │
│  actxcontrol(ProgID, . . .) │                    │  actxserver(ProgID, SysName)│
└─────────────────────────────┘                    └─────────────────────────────┘
              │                                                    │
              │                    ┌─────────────────────────────────┐
              │     DLL Component  │ COM API determines whether to   │
              │◄───────────────────│ use in-process or out-of-process│
              │                    │ server, based on component type │
              │                    └─────────────────────────────────┘
              │                                    │ EXE Component
              │                                    │
              │                          No      ╱ SysName ╲
              │                       ┌─────────◄  Parameter │
              │                       │         ╲ Specified? ╱
              │                       │           ╲        ╱
              │                       │            │ Yes
              │                       │            │
  ┌───────────────────┐  ┌───────────────────┐ ┌───────────────────┐
  │  Create in-process│  │ Create or connect │ │ Create or connect │
  │  server           │  │ to out-of-process │ │ to out-of-process │
  │                   │  │ server on local   │ │ server on remote  │
  │                   │  │ system            │ │ system            │
  └───────────────────┘  └───────────────────┘ └───────────────────┘
```

Get IUnknown interface from component

Attempt to get IDispatch interface from component

Success? — No → Return handle to IUnknown interface

Yes → Return handle to IDispatch interface

Creating an ActiveX Control

You can create an ActiveX control from the MATLAB client using either a graphical user interface (GUI) or the `actxcontrol` function from the command line. Either of these methods creates an instance of the control in the MATLAB client process and returns a handle to the primary interface to the COM object. Through this interface, you can access the object's public properties or methods. You can also establish additional interfaces to the

object, including interfaces that use IDispatch, and any custom interfaces that may exist.

This section describes how to create the control and how to position it in the MATLAB figure window.

- "Finding Out What Controls Are Installed" on page 9-4
- "Finding a Particular Control" on page 9-5
- "Creating Control Objects Using a GUI" on page 9-5
- "Creating Control Objects from the Command Line" on page 9-8
- "Repositioning the Control in a Figure Window" on page 9-9

Finding Out What Controls Are Installed

The `actxcontrollist` function shows you what COM controls are currently installed on your system. Type:

```
list = actxcontrollist
```

MATLAB displays a cell array listing each control, including its name, programmatic identifier (ProgID), and file name.

This example shows information that might be returned for several controls (your results might be different):

```
list = actxcontrollist;
s=sprintf(' Name = %s\n ProgID = %s\n File = %s\n', list{114:115,:})
```

MATLAB displays:

```
s =
 Name = OleInstall Class
 ProgID = Outlook Express Mime Editor
 File = OlePrn.OleInstall.1
 Name = OutlookExpress.MimeEdit.1
 ProgID = C:\WINNT\System32\oleprn.dll
 File = C:\WINNT\System32\inetcomm.dll
```

Finding a Particular Control

If you know the name of a control, you can find it in the list and display its ProgID and the path of the directory containing it. For example, the Mwsamp2 control is used in some of the examples in this documentation. You can find it with the following code:

```
list = actxcontrollist;
for ii = 1:length(list)
   if ~isempty(findstr('Mwsamp2',[list{ii,:}]))
      s = sprintf(' Name = %s\n ProgID = %s\n File = %s\n', ...
             list{ii,:})
   end
end
```

MATLAB displays:

```
s =
   Name = Mwsamp2 Control
   ProgID = MWSAMP.MwsampCtrl.2
   File =
   D:\Apps\MATLAB\R2006a\toolbox\matlab\winfun\win32\mwsamp2.ocx
```

The location of this file might be different on your system.

Creating Control Objects Using a GUI

Using the actxcontrolselect function is the simplest way to create a control object. This function displays a GUI listing all controls installed on your system. When you select an item from the list and click the **Create** button, MATLAB creates the control and returns a handle to it. Type:

```
h = actxcontrolselect
```

MATLAB displays:
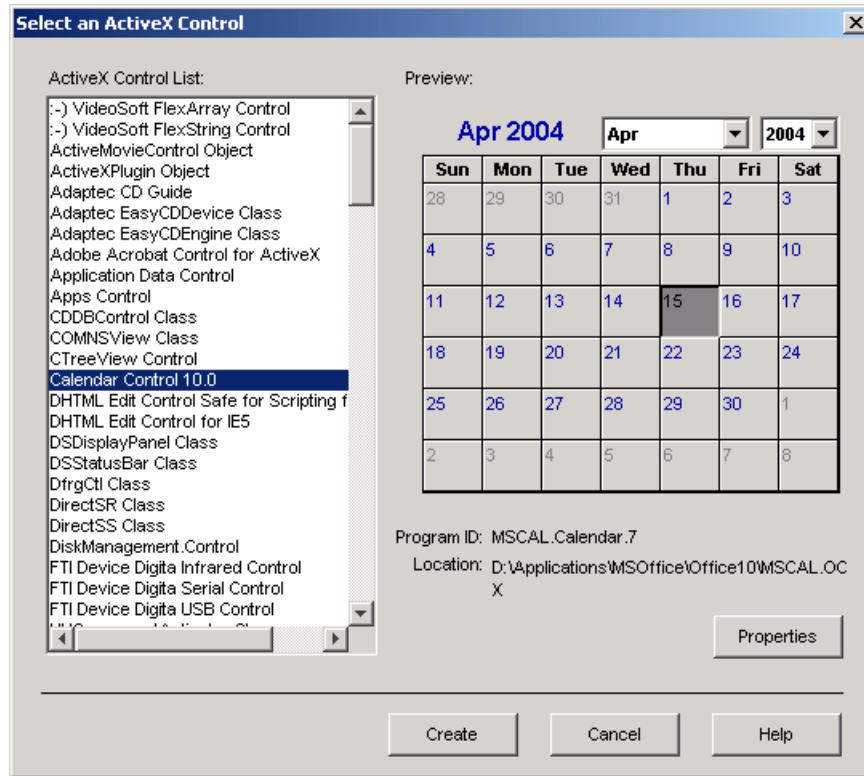
The interface has an **ActiveX Control List** selection pane on the left and a **Preview** pane on the right. Click one of the control names in the selection pane to see a preview of the control. (A blank preview pane means the control does not have a preview.) An error message appears in the preview pane if MATLAB cannot create the control.

Setting Properties with actxcontrolselect. Click the **Properties** button in the **Preview** pane to change property values when creating the control. You can select which figure window to put the control in (**Parent** field), where to position it in the window (**X** and **Y** fields), and what size to make the control (**Width** and **Height**).

You can register events you want the control to respond to in this window. (For an explanation of event registration, see "Responding to Events — an

Overview" on page 9-53.) Register an event and the callback routine to handle that event by entering the name of the routine to the right of the event under **Callback M-File**.

You can select callback routines by browsing for their M-files. Click a name in the **Event Name** column, and then click the **Browse** button. To assign a callback routine to more than one event, first press the **Ctrl** key and click individual event names, or drag the mouse over consecutive event names, and then click **Browse** to select the callback routine.

MATLAB only responds to registered events, so if you do not specify a callback M-File, the event is ignored.

For example, in the **ActiveX Control List** pane, select **Calendar Control 10.0** (the version on your system may be different) and click **Properties**. MATLAB displays the Choose ActiveX Control Creation Parameter dialog box. Enter a **Width** of 500 and a **Height** of 350 to change the default size for the control. Click **OK** in this window, and click **Create** in the next window to create the Calendar control.
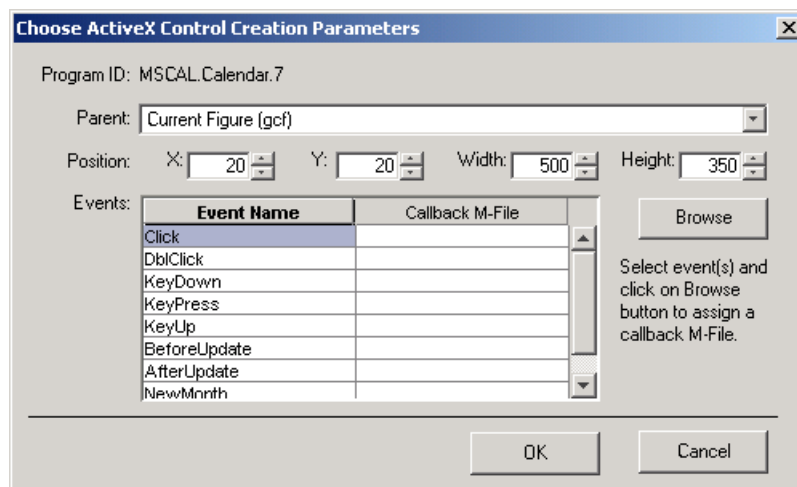
You can also set control parameters using the actxcontrol function. One parameter you can set with actxcontrol, but not with actxcontrolselect, is the name of an initialization file. When you specify this file name, MATLAB sets the initial state of the control to that of a previously saved control.

Information Returned by actxcontrolselect. The actxcontrolselect function creates an object that is an instance of the MATLAB COM class. The function returns up to two arguments: a handle for the object, h, and a 1-by-3 cell array, info, containing information about the control. To get this information, type:

```
[h, info] = actxcontrolselect
```

The cell array info shows the name, ProgID, and file name for the control.

If you select the Calendar Control, and then click **Create**, MATLAB displays information similar to:

```
h =
    COM.mscal.calendar.7
info =
    [1x20 char]    'MSCAL.Calendar.7'    [1x41 char]
```

To expand the info cell array, type:

```
info{:}
```

MATLAB displays:

```
ans =
    Calendar Control 9.0
ans =
    MSCAL.Calendar.7
ans =
    D:\Applications\MSOffice\Office\MSCAL.OCX
```

Creating Control Objects from the Command Line

If you already know which control you want and you know its ProgID, you can bypass the GUI by using the actxcontrol function to create it.

The ProgID is the only required input to this function. However, as with actxcontrolselect, you can supply additional inputs that enable you to select which figure window to put the control in, where to position it in the window, and what size to make it. You can also register any events you want the control to respond to, or set the initial state of the control by reading that

state from a file. See the `actxcontrol` reference page for a full explanation of its input arguments.

The `actxcontrol` function returns a handle to the primary interface to the object. Use this handle to reference the object in other COM function calls. You can also use the handle to obtain additional interfaces to the object. For more information on using interfaces, see "Getting Interfaces to the Object" on page 9-69.

This example creates a Microsoft Calendar control. Position the control in figure window `fig3`, at a `[0 0]` x-y offset from the bottom left of the window, and make it 300-by-400 pixels in size:

```
fig3 = figure('position', [50 50 600 500]);
h = actxcontrol('mscal.calendar', [0 0 300 400], fig3)
```

MATLAB displays:

```
h =
    COM.mscal.calendar
```

Repositioning the Control in a Figure Window

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

Observe what happens to the object created in the last section when you specify new origin coordinates (70, 120) and new width and height dimensions of 400 and 350:

```
h.move([70 120 400 350]);
```

Creating a COM Server

Instantiating a DLL Component

To create a server for a component implemented as a dynamic link library (DLL), use the `actxserver` function. MATLAB creates an instance of the component in the same process that contains the client application.

The syntax for `actxserver`, when used with a DLL component, is `actxserver(ProgID)`, where `ProgID` is the programmatic identifier for the component.

`actxserver` returns a handle to the primary interface to the object. Use this handle to reference the object in other COM function calls. You can also use the handle to obtain additional interfaces to the object. For more information on using interfaces, see "Getting Interfaces to the Object" on page 9-69.

Unlike Microsoft ActiveX controls, any user interface displayed by the server appears in a separate window.

You can not use a 32-bit in-process DLL COM object in a 64-bit MATLAB application. For information about this restriction, see the Technical Support solution 1-35LZ4G Why am I not able to use 32-bit DLL COM Objects in 64-bit MATLAB.

Instantiating an EXE Component

You can use the `actxserver` function to create a server for a component implemented as an executable (EXE). In this case, MATLAB instantiates the component in an out-of-process server.

The syntax for `actxserver`, when used to create an executable, is `actxserver(ProgID, sysname)`, where `ProgID` is the programmatic identifier for the component, and `sysname` is an optional argument used in configuring a distributed COM (DCOM) system.

`actxserver` returns a handle to the primary interface to the COM object. Use this handle to reference the object in other COM function calls. You can also use the handle to obtain additional interfaces to the object. For more information on using interfaces, see "Getting Interfaces to the Object" on page 9-69.

Any user interface displayed by the server appears in a separate window.

This example creates a COM server application running the Microsoft Excel spreadsheet program. The handle is assigned to `h`.

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

MATLAB displays:

```
h =
     COM.excel.application
```

MATLAB can programmatically connect to an instance of a COM Automation server application that is already running on your computer. Use the `actxGetRunningServer` function to get a reference to such an application. The syntax is `actxGetRunningServer(ProgID)`, where `ProgID` is the programmatic identifier for the component.

This example gets a reference to the Excel program, which must already be running on your system. The returned handle is assigned to `h`.

```
h = actxGetRunningServer('excel.application')
```

MATLAB displays:

```
h =
     COM.excel.application
```

Exploring Your Object

In this section...

About Your Object

A COM object has properties, methods, events, and interfaces. Your vendor documentation describes these features, but you can also learn about your object using MATLAB commands.

Exploring Properties

A *property* is information that is associated with a COM object. This topic shows you how to look at the properties of your object. For detailed information on reading and setting property values, see "Using Object Properties" on page 9-22.

To see a list of all properties of an object, you can use the get function or the Property Inspector, a GUI provided by MATLAB to display and modify properties.

In this section, we explore a Microsoft Excel object. To begin, create the object myApp:

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

Listing Properties

The get function lists all properties. For example, from the MATLAB command prompt, type:

```
myApp.get
```

MATLAB displays information similar to the following:

```
     Application: [1x1 Interface.Microsoft_Excel_9.0_
 Object_Library._Application]
         Creator: 'xlCreatorCode'
          Parent: [1x1 Interface.Microsoft_Excel_9.0_
 Object_Library._Application]
       ActiveCell: []
      ActiveChart: [1x50 char]
                  :
 OperatingSystem: 'Windows (32-bit) NT 5.01'
OrganizationName: 'The MathWorks'
                  :
```

One property is OrganizationName; its value in this example is The MathWorks.

Using the Property Inspector

The Property Inspector opens a new window showing the object's properties. This topic explains how to open it. For detailed information, see "Using the Property Inspector" on page 9-32.

You can open the Property Inspector using either of these methods:

- Call the inspect function from the MATLAB command line.

- Double-click the object in the MATLAB Workspace browser.

For example, type:

```
myApp.inspect
```

The Inspector window opens.
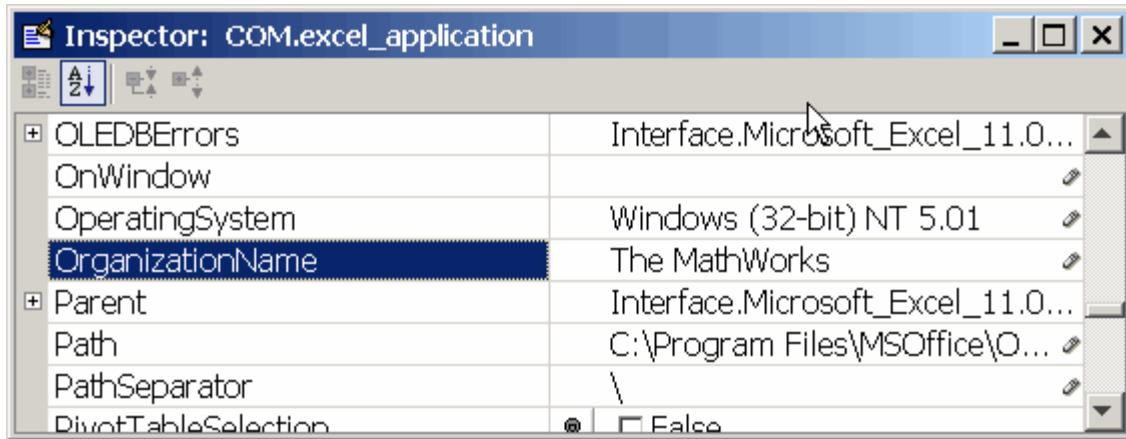
Scroll down until you see the `OrganizationName` property. It should be the same value the `get` function returned; in this case, `The MathWorks`.

Exploring Methods

A *method* is a procedure you call to perform a specific action on the COM object. This topic shows you how to identify methods belonging to your object. For detailed information, see "Using Methods" on page 9-39.

To see a list of all methods supported by an object, use the `methods` and `invoke` functions. Alternatively, you can use the `methodsview` function, which displays the methods in a separate window.

In this section, we explore a Microsoft Calendar object. To create the object `cal`, type:

```
cal = actxcontrol('mscal.calendar', [0 0 400 400]);
```

Listing Methods

The methods and invoke functions return a list of the names of all methods supported by the object, including MATLAB COM functions you can use on the object. For example, type:

```
cal.methods
```

MATLAB displays:

```
Methods for class COM.mscal_calendar:

AboutBox       PreviousMonth   constructorargs  invoke      send
NextDay        PreviousWeek    delete           load        set
NextMonth      PreviousYear    deleteproperty   move
NextWeek       Refresh         events           propedit
NextYear       Today           get              release
PreviousDay    addproperty     interfaces       save
```

When you use the -full switch, MATLAB also lists the input and output arguments for each method. For example, type:

```
cal.methods('-full')
```

MATLAB displays:

```
Methods for class COM.mscal.calendar:

HRESULT AboutBox(handle)
HRESULT NextDay(handle)
HRESULT NextMonth(handle)
HRESULT NextWeek(handle)
        :
MATLAB array move(handle, MATLAB array)
propedit(handle)
release(handle, MATLAB array)
save(handle, string)
```

The invoke function displays similar information for methods supported by the object. For example, type:

```
cal.invoke
```

MATLAB displays:

```
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)
```

Using methodsview

The methodsview function opens a new window with an easy-to-read display of all methods supported by the object, along with related fields of information, as described in the reference page. For example, type:

```
cal.methodsview
```

MATLAB displays:

Return Type	Name	Arguments	Inherited From
HRESULT	PreviousMonth	(handle)	COM.mscal_calendar
HRESULT	PreviousWeek	(handle)	COM.mscal_calendar
HRESULT	PreviousYear	(handle)	COM.mscal_calendar
HRESULT	Refresh	(handle)	COM.mscal_calendar
HRESULT	Today	(handle)	COM.mscal_calendar
	addproperty	(handle, string)	COM.mscal_calendar
MATLAB array	constructorargs	(handle)	COM.mscal_calendar
	delete	(handle, MATLAB array)	COM.mscal_calendar
	deleteproperty	(handle, string)	COM.mscal_calendar
MATLAB array	events	(handle, MATLAB array)	COM.mscal_calendar
MATLAB array	get	(handle)	COM.mscal_calendar
MATLAB array	get	(handle, MATLAB array, MATLAB array)	COM.mscal_calendar

Methods of class COM.mscal_calendar.release

If the **Return Type** field for a method is blank, the method returns `void`.

Exploring Events

An *event* is typically a user-initiated action that takes place in a server application, which often requires a reaction from the client. For example, a user clicking the mouse at a particular location in a server interface window might require the client take some action in response. When an event is *fired*, the server communicates this occurrence to the client. If the client is *listening* for this particular type of event, it responds by executing a routine called an *event handler*.

This topic shows you how to identify events available to your object. For detailed information, see "Using Events" on page 9-52. For information on event handlers, see "Writing Event Handlers" on page 9-64.

Use the `events` function to list all events known to the control or server and use the `eventlisteners` function to list only registered events.

In this section, we use the Microsoft Internet Explorer Web browser. To begin, create the object `myNet`:

```
myNet = actxserver('internetexplorer.application');
```

Listing Server Events

Type:

```
myNet.events
```

MATLAB displays event information similar to:

```
        :
StatusTextChange = void StatusTextChange(string Text)
ProgressChange = void ProgressChange(int32 Progress,int32 ProgressMax)
CommandStateChange = void CommandStateChange(int32 Command,bool Enable)
        :
```

Listing Registered Events

No events are registered at this time. If you type:

```
myNet.eventlisteners
```

MATLAB displays:

```
ans =
      {}
```

Exploring Interfaces

An *interface* is a set of related functions used to access a COM object's data. When you create a COM object using the `actxserver` or `actxcontrol` functions, MATLAB returns a handle to an interface. You use the `get` and `interfaces` functions to see other interfaces implemented by your object.

In this section, we explore an Excel object. To begin, create the object `e`:

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

Additional Interfaces

Components often provide additional interfaces, based on `IDispatch`. To see these interfaces, type:

```
e.get
```

MATLAB displays information similar to:

```
    Application: [1x1 Interface.Microsoft_Excel_11.0_Object_Library._Application]
        Creator: 'xlCreatorCode'
         Parent: [1x1 Interface.Microsoft_Excel_11.0_Object_Library._Application]
      ActiveCell: []
     ActiveChart: [1x50 char]
                :
       Workbooks: [1x1  Interface.Microsoft_Excel_11.0_Object_Library.Workbooks]
                :
```

In this example, `Workbooks` is an interface. To explore the `Workbooks` interface, type:

```
w = e.Workbooks;
```

To see its properties, type:

```
w.get
```

MATLAB displays:

```
    Application: [1x1 Interface.Microsoft_Excel_11.0_Object_Library._Application]
        Creator: 'xlCreatorCode'
         Parent: [1x1 Interface.Microsoft_Excel_11.0_Object_Library._Application]
          Count: 0
```

To see its methods, type:

```
w.invoke
```

MATLAB displays:

```
Add = handle Add(handle, Variant(Optional))
Close = void Close(handle)
Item = handle Item(handle, Variant)
Open = handle Open(handle, string, Variant(Optional))
OpenText = void OpenText(handle, string, Variant(Optional))
OpenDatabase = handle OpenDatabase(handle, string, Variant(Optional))
CheckOut = void CheckOut(handle, string)
CanCheckOut = bool CanCheckOut(handle, string)
OpenXML = handle OpenXML(handle, string, Variant(Optional))
```

Identifying Objects and Interfaces

You can get additional information about a control or server using the following functions.

Function	Description
class	Return the class of an object
isa	Determine if an object is of a given MATLAB class
iscom	Determine if the input is a COM or ActiveX object
isevent	Determine if an item is an event of a COM object

Function	Description
ismethod	Determine if an item is a method of a COM object
isprop	Determine if an item is a property of a COM object
isinterface	Determine if the input is a COM interface

This example creates a COM object in an Automation server running the Excel application, giving it the handle e, and a Workbooks interface to the object, with handle w.

```
e = actxserver('excel.application');
w = e.Workbooks;
```

Use the iscom function to see if e is a handle to a COM object:

```
e.iscom
ans =
     1
```

Use the isa function to test e against a known class name:

```
e.isa('COM.excel_application')
ans =
     1
```

Use isinterface to see if w is a handle to a COM interface:

```
w.isinterface
ans =
     1
```

Use the class function to find out the class of variable w:

```
w.class
ans =
    Interface.Microsoft_Excel_11.0_Object_Library.Workbooks
```

To see if UsableWidth is a property of e, use isprop:

```
e.isprop('UsableWidth')
ans =
```

     1

To see if `SaveWorkspace` is a method of `e`, use `ismethod`:

```
e.ismethod('SaveWorkspace')
ans =
     1
```

Using Object Properties

In this section...
"About Object Properties" on page 9-22
"Working with Properties" on page 9-23
"Setting the Value of a Property" on page 9-26
"Working with Multiple Objects" on page 9-28
"Using Enumerated Values for Properties" on page 9-29
"Using the Property Inspector" on page 9-32
"Custom Properties" on page 9-34
"Properties That Take Arguments" on page 9-35

About Object Properties

You can get the value of a property, and, in some cases, change the value. You also can create custom properties. This topic explains how to do these tasks. If you only want to view your object's properties, see "Exploring Properties" on page 9-12 for basic information.

Property names are not case sensitive. They can be abbreviated, as long as the name is unambiguous.

Use these MATLAB functions to work with the properties of a COM object.

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
isprop	Determine if an item is a property of a COM object

Function	Description
propedit	Display the control's built-in property page
set	Set the value of one or more properties

In this topic, you can use Microsoft Calendar control to demonstrate these functions. To begin, create the calendar object `cal`. A figure window opens; leave it open as you try the examples in this topic. Type:

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

Working with Properties

This section covers the following topics:

- "Listing Properties and Interfaces" on page 9-23

- "Getting Property Values" on page 9-24

- "Abbreviating Property Names" on page 9-25

- "Getting Multiple Property Values" on page 9-25

- "Working with Interfaces" on page 9-25

Listing Properties and Interfaces

The `get` function lists all properties and interfaces of the object. The `inspect` function opens the Property Inspector, which is described in "Using the Property Inspector" on page 9-32.

Using the previously created calendar object, type:

```
cal.get
```

MATLAB displays a list of all available properties and interfaces (the values for your object will be different):

```
        BackColor: 2147483663
              Day: 13
          DayFont: [1x1 Interface.Microsoft_Forms_2.0_Object_Library.Font]
     DayFontColor: 0
        DayLength: 1
```

```
              FirstDay: 7
          GridCellEffect: 1
               GridFont: [1x1 Interface.Microsoft_Forms_2.0_Object_Library.Font]
          GridFontColor: 10485760
          GridLinesColor: 2147483664
                  Month: 8
            MonthLength: 1
      ShowDateSelectors: 1
               ShowDays: 1
    ShowHorizontalGrid: 1
              ShowTitle: 1
       ShowVerticalGrid: 1
               TitleFont: [1x1 Interface.Microsoft_Forms_2.0_Object_Library.Font]
          TitleFontColor: 10485760
                  Value: '8/13/2007'
             ValueIsNull: O
                   Year: 2007
```

Getting Property Values

In this example, Year is a property and TitleFont is an interface. For information about interfaces, see "Working with Interfaces" on page 9-25. The following table shows different ways to get the value of the Year property.

Command	Description
myYear = cal.Year	Use dot syntax
myYear = cal.get('Year')	Use the get function
myYear = cal.year	Property names are not case sensitive
myYear = cal.ye	Property names can be abbreviated

MATLAB displays the same value for each of these commands, for example:

```
myYear =
        2007
```

Abbreviating Property Names

You can abbreviate property names, as long as the name is unambiguous.

Using the previously created calendar object `cal`, the command `cal.showda` is ambiguous because MATLAB cannot distinguish between the properties `ShowDateSelectors` and `ShowDays`. The command `cal.showdat` is unambiguous.

Getting Multiple Property Values

To get values for more than one property using a single command, you must use the `get` function. The values are returned a cell array. The syntax of this command is:

```
C = h.get({'prop1', 'prop2', ...});
```

Using the previously created calendar object, type:

```
myDate = cal.get({'Day','Month','Year'});
myDate{:}
```

MATLAB displays the current date, for example:

```
ans =
     13

ans =
      8

ans =
    2007
```

Working with Interfaces

The `TitleFont` interface provides additional functionality for your calendar object. To work with this interface, create a calendar title object `calTitle` then list its properties. For example, type:

```
calTitle=cal.TitleFont;
calTitle.get
```

MATLAB displays the available properties and their current values. For example:

```
          Name: 'Arial'
          Size: 12
          Bold: 1
        Italic: 0
     Underline: 0
 Strikethrough: 0
        Weight: 700
       Charset: 0
```

Setting the Value of a Property

This section covers the following topics:

- "Command Line Options" on page 9-26

- "Setting Multiple Property Values" on page 9-27

- "Setting Values with the Property Inspector" on page 9-27

- "Using the Property Page" on page 9-27

- "Using the Control GUI" on page 9-28

Command Line Options

You can set property values from the command line using different syntax statements. Working with the previously defined `calTitle` object, select your calendar figure window and observe the month name as you type the following commands.

Command	Description
`calTitle.Size=30;`	Use dot syntax
`calTitle.Name='Times New Roman';`	Use the set function
`calTitle.italic=1;`	Property names are not case sensitive
`calTitle.set('Under',1);`	Property names can be abbreviated

After making these changes, type:

```
calTitle.get
```

MATLAB displays the updated values:

```
          Name: 'Times New Roman'
          Size: 30
          Bold: 1
        Italic: 1
     Underline: 1
 Strikethrough: 0
        Weight: 700
       Charset: 0
```

Setting Multiple Property Values

To change more than one property with one command, you must use the set function. The syntax of this command is:

```
handle.set('pname1', value1, 'pname2', value2, ...)
```

For example, observe the month name in your calendar figure window when you type:

```
calTitle.set('Size',9,'Underline',0,'Italic',0);
```

Setting Values with the Property Inspector

You can use the Property Inspector to change values. For information, see "Using the Property Inspector" on page 9-32.

Using the Property Page

Some controls have a built-in property page. The propedit function gives you access to this page. You can both read and set property values. For example, typing:

```
cal.propedit
```

opens the **ActiveX Control Properties** window. You can experiment with changing values. To save new values, click the **Apply** button. Depending on what changes you make, type `cal.get` or `cal.titlefont.get` to see the new values.

Using the Control GUI

The Microsoft Calendar control provides a GUI for changing values. Select your calendar figure window and observe as you change the month and year from the drop-down lists, and click on a day of the month. To see your changes, at the command line type:

```
cal.Value
```

MATLAB displays the updated date, for example:

```
ans =

3/15/2008
```

Working with Multiple Objects

You can use the `get` and `set` functions on more than one object at a time by creating a vector of object handles and using these commands on the vector.

This example creates a vector `H` of handles to four calendar objects.

```
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];
```

Click on different days on each of the calendars. To see your changes, type:

```
H.get('Day')
```

MATLAB displays the day for each calendar. For example:

```
ans =
    [20]
    [18]
```

```
     [ 8]
     [29]
```

To change the `Day` on all four calendars, type:

```
H.set('Day', 23)
```

To see the results, type:

```
H.get('Day')
```

MATLAB displays:

```
ans =
     [23]
     [23]
     [23]
     [23]
```

Note To get or set values for multiple objects, you must use the `get` and `set` functions explicitly. You can only use dot syntax, for example `H.Day`, on scalar objects.

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 a Microsoft Excel spreadsheet is `xlUnicodeText`. This is easier to remember than a numeric value like 57.

This section covers the following topics:

- "Finding All Enumerated Properties" on page 9-30

- "Setting Enumerated Values" on page 9-31

- "Setting Enumerated Values with the Property Inspector" on page 9-32

Finding All Enumerated Properties

The `get` and `set` functions support enumerated values for properties for those applications that provide them. Use the `set` function to see which properties use enumerated types.

For example, create an Excel spreadsheet:

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

Type:

```
h.set
```

MATLAB displays:

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

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

Use the `get` function to display the current values of these properties. Type:

```
h.get
```

MATLAB displays information such as:

```
                Creator: 'xlCreatorCode'
       ConstrainNumeric: 0
  CopyObjectsWithCells: 1
                 Cursor: 'xlDefault'
             CutCopyMode: ''
                          .
                          .
```

Setting Enumerated Values

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:

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

To set the value of a property, assign the enumerated value to the property name:

```
handle.property = 'enumeratedvalue';
```

You can also use the `set` function with the property name and enumerated value:

```
handle.set('property', 'enumeratedvalue');
```

You have a choice of using the enumeration or its equivalent numeric value. You can abbreviate the enumeration string, as in the third line of the following example, as long as you use enough letters in the string to make it unambiguous. Enumeration strings are not case sensitive.

Make the Excel spreadsheet window visible, and then change the cursor from the MATLAB client. To see how the cursor has changed, you need to click the spreadsheet window. Either of the following assignments to `h.Cursor` sets the cursor to the `Wait` (hourglass) type:

```
h.Visible = 1;

h.Cursor = 'xlWait'
h.Cursor = 'xlw'              % Abbreviated form of xlWait
```

Read the value of the `Cursor` property you have just set:

```
h.Cursor
ans =
   xlWait
```

Setting Enumerated Values with the Property Inspector

You can also set enumerated values using the Property Inspector. To learn how to use this feature, see "Using the Property Inspector on Enumerated Values" on page 9-33.

Using the Property Inspector

The Property Inspector enables you to access the properties of COM objects. To open the Property Inspector, use the inspect function from the MATLAB command line or double-click the object in the MATLAB Workspace browser.

For example, create a server object running the Excel program. Then set the object's DefaultFilePath property to C:\ExcelWork:

```
h = actxserver('excel.application');
h.DefaultFilePath = 'C:\ExcelWork';
```

Next call the inspect function to display a new window showing the server object's properties:

```
h.inspect
```

Scroll down until you see the DefaultFilePath property that you just changed. It should read C:\ExcelWork.
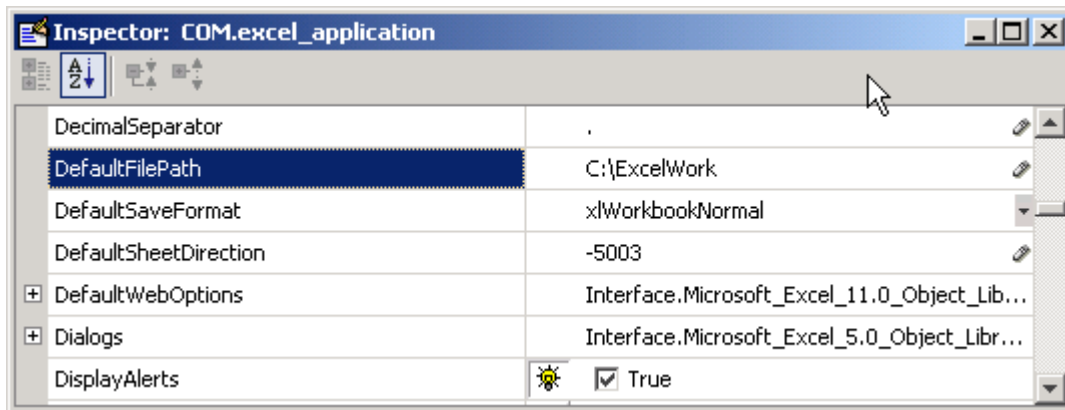
Using the Property Inspector, change `DefaultFilePath` once more, this time to `MyWorkDirectory`. To do this, select the value at the right and type the new value.

Now go back to the MATLAB Command Window and confirm that the `DefaultFilePath` property has changed as expected.

```
h.DefaultFilePath
```

MATLAB displays:

```
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 window by reinvoking `inspect` on the object.

Using the Property Inspector on Enumerated Values

A list button next to a property value indicates the property accepts enumerated values. Click anywhere in the field on the right to see the values. The following figure displays four enumerated values for the `Cursor` property. The current value `xlDefault` is displayed in the field next to the property name.
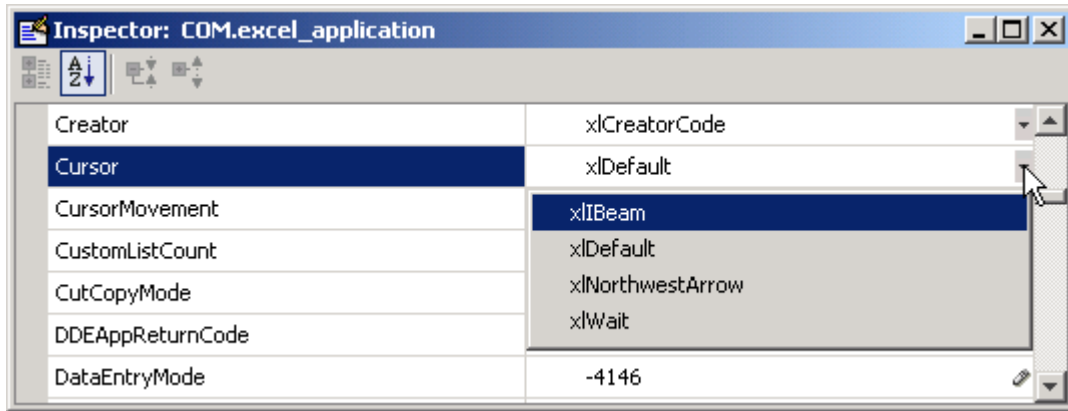
To change the value, use the list button to display the options for that property, and then click the desired value.

Custom Properties

You can create your own properties to a control using the addproperty function. The syntax h.addproperty('propertyName') creates a custom property for control h.

This example creates the mwsamp2 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]);
h.addproperty('Position');
h.Position = [200 120];
```

Use the get function to list all properties of control h.

```
h.get
```

You see the new Position property has been added.

```
ans =
      Label: 'Label'
     Radius: 20
   Position: [200 120]
```

Type:

```
h.Position
```

MATLAB displays:

```
ans =
    200    120
```

To remove custom properties from a control, use the `deleteproperty` function. The syntax `h.deleteproperty('propertyName')` deletes `propertyName` from `h`. For example, to delete the `Position` property that you just created and show that it no longer exists, type:

```
h.deleteproperty('Position');
h.get
```

MATLAB displays:

```
ans =
     Label: 'Label'
    Radius: 20
```

Properties That Take Arguments

Some COM objects have properties that accept input arguments. Internally, MATLAB handles these properties as methods, which means you need to use the `invoke` function (not `get`) to view the property.

To explain how this works, look at a spreadsheet property that takes input arguments. This example is taken from "Using a MATLAB Application as an Automation Client" on page 9-84.

- "An Example" on page 9-36
- "Exploring the Object" on page 9-36
- "Exploring Values" on page 9-36
- "Setting Values" on page 9-38
- "Completing the Example" on page 9-38

An Example

The Excel Activesheet interface is an object that takes input arguments. This interface has a property called Range. To specify Range, you must pass in range coordinates.

To begin, create the Worksheet object ws:

```
e = actxserver('excel.application');
e.Workbooks.Add;
ws = e.Activesheet;
```

The ws object is an interface:

```
ws =
    Interface.Microsoft_Excel_11.O_Object_Library._Worksheet
```

Exploring the Object

You can explore the ws object using the get and invoke functions. (When you type the following commands, MATLAB displays long lists of properties and methods.) When you type ws.get, the property Range is not in the list. You must use the invoke function to find Range.

```
ws.invoke
```

MATLAB displays (in part):

```
              :
Range = handle Range(handle, Variant, Variant(Optional))
              :
```

Exploring Values

The get function also displays the value of a property. For example, one of the properties listed by get is StandardHeight. To see its value, type:

```
ws.get('StandardHeight')
```

MATLAB displays:

```
ans =
    13.2000
```

But, if you use this command on `Range`:

```
ws.get('Range');
```

MATLAB displays:

```
??? Invoke Error: Incorrect number of arguments
```

Consulting Microsoft reference documentation, you find `Range` requires arguments `A1:B2`, which specify a rectangular region of the spreadsheet.

If you type:

```
wsRange = ws.get('Range', 'A1:B2')
```

MATLAB shows that `wsRange` is an interface:

```
wsRange =
    Interface.Microsoft_Excel_11.0_Object_Library.Range
```

You find the properties by typing:

```
wsRange.get
```

From the lengthy list MATLAB displays, look at the `Value` property:

```
            :
    Value: {2x2 cell}
            :
```

To see the current value, type:

```
wsRange.Value
```

MATLAB displays:

```
ans =
    [NaN]    [NaN]
    [NaN]    [NaN]
```

Setting Values

To copy a MATLAB array A into the wsRange object, type:

```
A = [1 2; 3 4];
wsRange.Value = A;
wsRange.Value
```

MATLAB displays:

```
ans =
    [1]    [2]
    [3]    [4]
```

Completing the Example

When you are finished with this example, type:

```
e.Workbook.Close;
```

The Excel Close method expects a Yes/No response about saving the workbook. To terminate and remove the server object, type:

```
e.Quit;
e.delete;
```

Using Methods

About Methods

You execute, or *invoke*, COM functions or methods belonging to COM objects. This topic explains how to determine what methods are available for an object and how to invoke them. If you only want to view your object's methods, see "Exploring Methods" on page 9-14 for basic information.

Method names are case sensitive. You cannot abbreviate them.

Use the following MATLAB functions to work with the methods of a COM object.

Function	Description
invoke	Invoke a method or display a list of methods and types
ismethod	Determine if an item is a method of a COM object
methods	List all method names for the control or server
methodsview	Graphic display of information on all methods and types

Getting Method Information

You can see what methods are supported by a COM object using the `methodsview`, `methods`, or `invoke` functions. Each function presents specific information, as described in the following table.

Function	Output
`invoke`	Cell array of function names and signatures
`methods`	Cell array of function names only, sorted alphabetically, with uppercase names listed first
`methods` with `-full` qualifier	Cell array of function names and signatures, sorted alphabetically
`methodsview`	Graphical display of function names and signatures

In this topic, you can use the built-in MATLAB control `mwsamp` to try out these functions. To create the control object `sampObj`, type:

```
sampObj = actxcontrol('mwsamp.mwsampctrl.1', [0 0 500 500]);
```

The control opens a figure window and displays a circle and text label.

Using invoke

The `invoke` function returns a cell array containing a list of all methods supported by the object, along with the signatures for these methods. This list is not sorted alphabetically.

For example, type:

```
sampObj.invoke
```

MATLAB displays:

```
Beep = void Beep(handle)
Redraw = void Redraw(handle)
GetVariantArray = Variant GetVariantArray(handle)
GetIDispatch = handle GetIDispatch(handle)
```

```
GetBSTR = string GetBSTR(handle)
GetI4Array = Variant GetI4Array(handle)
GetBSTRArray = Variant GetBSTRArray(handle)
GetI4 = int32 GetI4(handle)
GetR8 = double GetR8(handle)
GetR8Array = Variant GetR8Array(handle)
FireClickEvent = void FireClickEvent(handle)
GetVariantVector = Variant GetVariantVector(handle)
GetR8Vector = Variant GetR8Vector(handle)
GetI4Vector = Variant GetI4Vector(handle)
SetBSTRArray = Variant SetBSTRArray(handle, Variant)
SetI4 = int32 SetI4(handle, int32)
SetI4Vector = Variant SetI4Vector(handle, Variant)
SetI4Array = Variant SetI4Array(handle, Variant)
SetR8 = double SetR8(handle, double)
SetR8Vector = Variant SetR8Vector(handle, Variant)
SetR8Array = Variant SetR8Array(handle, Variant)
SetBSTR = string SetBSTR(handle, string)
AboutBox = void AboutBox(handle)
```

Using methods

The methods function returns the names of all methods for the object, including MATLAB COM functions that you can use on the object. There is no information about how to call the method. This list is sorted alphabetically; however, method names with initial caps are listed before methods with lowercase names.

For example, type:

```
sampObj.methods
```

MATLAB displays:

```
Methods for class COM.mwsamp_mwsampctrl_1:

AboutBox         GetVariantVector  deleteproperty
Beep             Redraw            events
FireClickEvent   SetBSTR           get
GetBSTR          SetBSTRArray      interfaces
```

```
GetBSTRArray      SetI4             invoke
GetI4             SetI4Array        load
GetI4Array        SetI4Vector       move
GetI4Vector       SetR8             propedit
GetIDispatch      SetR8Array        release
GetR8             SetR8Vector       save
GetR8Array        addproperty       send
GetR8Vector       constructorargs   set
GetVariantArray   delete
```

Examples of MATLAB COM functions are `addproperty` and `set`. Although the list is sorted alphabetically, uppercase function names are listed first. For example, `Redraw` appears before `get`.

Using methods with -full

When you include the `-full` qualifier in the `methods` function, MATLAB also specifies the input and output arguments for each method. For an overloaded method, the returned array includes a description of each of its signatures.

Type:

```
sampObj.methods('-full')
```

MATLAB displays:

```
Methods for class COM.mwsamp_mwsampctrl_1:

AboutBox(handle)
Beep(handle)
FireClickEvent(handle)
string GetBSTR(handle)
Variant GetBSTRArray(handle)
int32 GetI4(handle)
Variant GetI4Array(handle)
Variant GetI4Vector(handle)
handle GetIDispatch(handle)
double GetR8(handle)
Variant GetR8Array(handle)
Variant GetR8Vector(handle)
Variant GetVariantArray(handle)
```

```
Variant GetVariantVector(handle)
Redraw(handle)
string SetBSTR(handle, string)
Variant SetBSTRArray(handle, Variant)
int32 SetI4(handle, int32)
Variant SetI4Array(handle, Variant)
Variant SetI4Vector(handle, Variant)
double SetR8(handle, double)
Variant SetR8Array(handle, Variant)
Variant SetR8Vector(handle, Variant)
addproperty(handle, string)
MATLAB array constructorargs(handle)
delete(handle, MATLAB array)
deleteproperty(handle, string)
MATLAB array events(handle, MATLAB array)
MATLAB array get(handle)
MATLAB array get(handle, MATLAB array, MATLAB array)
MATLAB array get(handle vector, MATLAB array, MATLAB array)
MATLAB array interfaces(handle)
MATLAB array invoke(handle)
MATLAB array invoke(handle, string, MATLAB array)
load(handle, string)
MATLAB array move(handle, MATLAB array)
MATLAB array move(handle)
propedit(handle)
release(handle, MATLAB array)
save(handle, string)
MATLAB array send(handle)
MATLAB array set(handle vector, MATLAB array, MATLAB array)
MATLAB array set(handle, MATLAB array, MATLAB array)
MATLAB array set(handle)
```

In the mwsamp control, get is an overloaded function, and MATLAB displays each of its signatures.

Using methodsview

The methodsview function opens a new window with an easy-to-read display of all methods supported by the object. It displays the same information as the *handle*.methods('-full') command.

For example, type:

```
sampObj.methodsview
```

MATLAB opens a window showing (in part):

Return Type	Name	Arguments	Inherited From
	Redraw	(handle)	COM.mwsamp_mwsampctrl_1
string	SetBSTR	(handle, string)	COM.mwsamp_mwsampctrl_1
Variant	SetBSTRArray	(handle, Variant)	COM.mwsamp_mwsampctrl_1
int32	SetI4	(handle, int32)	COM.mwsamp_mwsampctrl_1
Variant	SetI4Array	(handle, Variant)	COM.mwsamp_mwsampctrl_1
Variant	SetI4Vector	(handle, Variant)	COM.mwsamp_mwsampctrl_1
double	SetR8	(handle, double)	COM.mwsamp_mwsampctrl_1
Variant	SetR8Array	(handle, Variant)	COM.mwsamp_mwsampctrl_1
Variant	SetR8Vector	(handle, Variant)	COM.mwsamp_mwsampctrl_1
	addproperty	(handle, string)	COM.mwsamp_mwsampctrl_1
MATLAB array	constructorargs	(handle)	COM.mwsamp_mwsampctrl_1
	delete	(handle, MATLAB array)	COM.mwsamp_mwsampctrl_1
	deleteproperty	(handle, string)	COM.mwsamp_mwsampctrl_1

Title bar: **Methods of class COM.mwsamp_mwsa...**

Invoking Methods on an Object

This section covers the following topics:

- "Calling Syntax" on page 9-44
- "Input and Output Arguments" on page 9-45
- "Example Using mwsamp" on page 9-45

Calling Syntax

To invoke a method on a COM object, use *dot syntax*, also called dot notation. This is a simpler syntax that doesn't require an explicit function call. For situations where you cannot use this syntax, see "Exceptions to Using Implicit Syntax" on page 9-46.

The format of a dot syntax statement is:

```
outputvalue = object.methodname('arg1', 'arg2', ...);
```

Specify the object name, the dot (.), and the name of the function or method. Enclose any input arguments in parentheses after the function name. Specify output arguments to the left of the equal sign.

Dot syntax is a special case of calling by method name. An alternative syntax for calling by method name is:

```
outputvalue = methodname(object, 'arg1', 'arg2', ...);
```

MATLAB also supports the following explicit syntax statements:

```
outputvalue = invoke(object, 'methodname', 'arg1', 'arg2', ...);
outputvalue = object.invoke('methodname', 'arg1', 'arg2', ...);
```

Input and Output Arguments

The `methodsview` output window and the `methods -full` command show what data types to use for input and output arguments. For information about reading a signature statement and using input and output arguments, see "Handling COM Data in MATLAB Software" on page 9-74.

Example Using mwsamp

The following example creates three circles in a MATLAB figure window. It shows different commands you can use to change the circles.

To create the COM objects, type:

```
h1 = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200]);
h2 = actxcontrol('mwsamp.mwsampctrl.2', [200 200 200 200]);
h3 = actxcontrol('mwsamp.mwsampctrl.2', [400 0 200 200]);
```

You can explicitly change the size of and redraw a circle using the commands:

```
h1.set('Radius', 100);
invoke(h1, 'Redraw')
```

You can implicitly change the size using:

```
h2.Radius = 50;
h3.Radius = 25;
```

To redraw the circles using method name syntax, type:

```
Redraw(h2)
h3.Redraw
```

Close the figure window.

Exceptions to Using Implicit Syntax

You cannot use dot syntax and must explicitly call the `get`, `set`, and `invoke` functions under the following conditions:

- "Accessing Nonpublic Properties and Methods" on page 9-46
- "Accessing Properties That Take Arguments" on page 9-47
- "Operating on a Vector of Objects" on page 9-47

Accessing Nonpublic Properties and Methods

If the property or method you want to access is not a public property or method of the object class, or if it is not in the type library for the control or server, you must call `get`, `set`, or `invoke` explicitly.

If you use a syntax statement of the following format for a nonpublic property *aProperty*:

```
x = handle.aProperty
```

MATLAB displays a message such as:

```
??? No appropriate method or public field aProperty for class
    COM.aClass.application.
```

Instead, you must use the `get` function explicitly:

```
x = handle.get('aProperty')
```

To find public properties and methods on COM object h, type:

```
publicproperties = h.get
publicmethods = h.invoke
```

Accessing Properties That Take Arguments

Some COM objects have properties that accept input arguments. MATLAB treats these properties like methods. For an example of this feature, see "Properties That Take Arguments" on page 9-35.

To get or set the value of such a property, you must make an explicit call to the `get` or `set` function, as shown in the following example. In this example, `A1` and `B2` are arguments that specify which `Range` interface to return on the `get` operation:

```
eActivesheetRange = e.Activesheet.get('Range', 'A1', 'B2');
```

Operating on a Vector of Objects

If you operate on a vector of objects you must call `get` or `set` explicitly to access properties. For an example, see "Working with Multiple Objects" on page 9-28. This applies only to the `get` and `set` functions. You cannot invoke a method on multiple COM objects, even if you call the `invoke` function explicitly.

This example creates a vector of handles to two Microsoft Calendar objects. It then modifies the `Day` property of both objects in one operation by invoking `set` on the vector, as follows:

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

Observe the figure window as you type:

```
H.set('Day', 23)
```

To verify, type:

```
H.get('Day')
```

MATLAB displays:

```
ans =
    [23]
    [23]
```

Close the figure window.

Specifying Enumerated Parameters

Enumeration is a way of assigning a descriptive name to a symbolic value.

For example, the input to a function is the atomic number of an element. It is easier to remember an element name than the atomic number. Using enumeration, you can pass the word `'arsenic'` in place of the value 33.

MATLAB supports enumeration for parameters passed to methods under the condition that the type library in use reports the parameter as `ENUM`, and only as `ENUM`.

Note MATLAB does not support enumeration for any parameter that the type library reports as both `ENUM` and `Optional`.

In this example, the `Location` method accepts the enumerated value `'xlLocationAsObject'`.

Create a Microsoft Excel `Chart` object:

```
e = actxserver('Excel.Application');

% Insert a new workbook.
Workbook = e.Workbooks.Add;
e.Visible = 1;
Sheets = e.ActiveWorkBook.Sheets;

% Get a handle to the active sheet.
Activesheet = e.Activesheet;

%Add a Chart
Charts = Workbook.Charts;
```

```
Chart = Charts.Add;
```

To see what type of chart you can create, type:

```
Chart.inspect
```

Scroll through the Property Inspector window to find `ChartType`. Click the drop-down arrow to see all possible `ChartType` values. This is an enumerated list. Close the property inspector.

To programmatically set the `ChartType`, type:

```
% Set chart type to be a line plot.
Chart.ChartType = 'xlXYScatterLines'
C1 = Chart.Location('xlLocationAsObject', 'Sheet1');
```

Close the Excel spreadsheet.

Optional Input Arguments

When calling a method that takes optional input arguments, you can skip any optional argument by specifying an empty array (`[]`) in its place. The syntax for calling a method with second argument `arg2` not specified is:

```
handle.methodname(arg1, [], arg3);
```

The following example uses the `Add` method to add new sheets to an Excel workbook. The `Add` method has the following optional input arguments:

- `Before` — The sheet before which to add the new sheet
- `After` — The sheet after which to add the new sheet
- `Count` — The total number of sheets to add
- `Type` — The type of sheet to add

The following code creates a workbook with the default number of worksheets, and inserts an additional sheet after Sheet 1. To do this, call `Add` with the second argument, `After`. You omit the first argument, `Before`, by using `[]` in its place, as shown in the last line of the example:

```
% Open an Excel Server.
```

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

% Insert a new workbook.
e.Workbooks.Add;
e.Visible = 1;

% Get the Active Workbook with three sheets.
eSheets = e.ActiveWorkbook.Sheets;

% Add a new sheet after eSheet1.
eSheet1 = eSheets.Item(1);
eNewSheet = eSheets.Add([], eSheet1);
```

Close the Excel spreadsheet.

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.

The following syntax shows a server function functionname called by the MATLAB client. retval is the function's first output argument, or return value. The other output arguments are out1, out2, ...

```
[retval out1 out2 ...] = handle.functionname(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; MATLAB does not support pass-by-reference.

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 the following, identifying the invalid argument.

```
??? Error: Type mismatch, argument 3.
```

If you do not use the dot syntax format, be careful interpreting the argument number in this type of message.

For example, using dot syntax, if you type:

```
handle.PutFullMatrix('a', 'base', 7, [5 8]);
```

MATLAB displays:

```
??? Error: Type mismatch, argument 3.
```

In this case, the argument, 7, is invalid because `PutFullMatrix` expects the third argument to be an array data type, not a scalar. In this example, the error message identifies 7 as `argument 3`.

However, if you use the syntax:

```
PutFullMatrix(handle, 'a', 'base', 7, [5 8]);
```

MATLAB displays:

```
??? Error: Type mismatch, argument 3.
```

In this call to the `PutFullMatrix` function, 7 is argument four. However, the COM server does not receive the first argument. The `handle` argument merely identifies the server. It does not get passed to the server. This means the server sees `'a'` as the first argument, and the invalid argument, 7, as the third.

If you use the syntax:

```
invoke(handle, 'PutFullMatrix', 'a', 'base', 7, [5 8]);
```

MATLAB again displays:

```
??? Error: Type mismatch, argument 3.
```

As in the previous example, MATLAB uses the `handle` argument to identify the server. The `'PutFullMatrix'` argument is also only used by MATLAB. While the invalid argument is the fifth argument in your MATLAB command, the server still identifies it as argument 3, because the first two arguments are not seen by the server.

Using Events

In this section...

About Events

An *event* is typically a user-initiated action that takes place in a server application, which often requires a reaction from the client. For example, a user clicking the mouse at a particular location in a server interface window might require the client take some action in response. When an event is *fired*, the server communicates this occurrence to the client. If the client is *listening* for this particular type of event, it responds by executing a routine called an *event handler*.

The MATLAB COM client can subscribe to and handle the events fired by a Microsoft ActiveX control or a COM server. Select the events you want the client to listen to by registering each event you want active with the event handler to be used in responding to the event. When a registered event takes place, the control or server notifies the client, which responds by executing the appropriate event handler routine. You can write M-files for event handlers.

Note MATLAB does not support interface events from a Custom server.

Functions for Working with Events

Use the MATLAB functions in the following table to register and unregister events, to list all events, or to list just registered events for a control or server.

Function	Description
actxcontrol	Create a COM control and optionally register those events you want the client to listen to
eventlisteners	Return a list of events attached to listeners
events	List all events, both registered and unregistered, a control or server can generate
isevent	Determine if an item is an event of a COM object
registerevent	Register an event handler with a control or server event
unregisterallevents	Unregister all events for a control or server
unregisterevent	Unregister an event handler with a control or server event

Event names and event handler names are not case sensitive. You cannot abbreviate them.

Examples of Event Handlers

The following examples use event handlers:

- "Example — Grid ActiveX Control in a Figure" on page 8-16
- "Example — Reading Excel Spreadsheet Data" on page 8-24

Responding to Events — an Overview

This section describes the basic steps to handle events fired by a COM control or server.

- "Identifying All Events" on page 9-54
- "Registering Those Events You Want to Respond To" on page 9-54

Identifying All Events

Use the `events` function to list all events the control or server can respond to. This function returns a structure array, where each field of the structure is the name of an event handler, and the value of that field contains the signature for the handler routine. To invoke `events` on an object with handle `h`, type:

```
S = h.events
```

Registering Those Events You Want to Respond To

Use the `registerevent` function to register those server events you want the client to respond to. You can register events as follows:

- If you have one function to handle all server events, register this common event handler using the syntax:

```
h.registerevent('handler');
```

- If you have a separate event handler function for different events, use the syntax:

```
h.registerevent({'event1' 'handler1'; 'event2' 'handler2';
...});
```

For ActiveX controls, you can register events at the time you create the control using the `actxcontrol` function.

- To register a common event handler function to respond to all events, use:

```
h = actxcontrol('progid', position, figure, 'handler');
```

- To register a separate function to handle each type of event, use:

```
h = actxcontrol('progid', position, figure, ...
    {'event1' 'handler1'; 'event2' 'handler2'; ...});
```

The MATLAB client responds only to events you have registered.

Identifying Registered Events

The `eventlisteners` function lists only currently registered events. This function returns a cell array, with each row representing a registered event and the name of its event handler. For example, to invoke `eventlisteners` on an object with handle `h`, type:

```
C = h.eventlisteners
```

Responding to Events As They Occur

Whenever a control or server fires an event that the client is listening for, the client responds to the event by invoking one or more event handlers that have been registered for that event. You can implement these routines in M-file programs that you write to handle events. Read more about event handlers in the section on "Writing Event Handlers" on page 9-64.

Unregistering Events You No Longer Want to Listen To

If you have registered events that you now want the client to ignore, you can unregister them at any time using the functions `unregisterevent` and `unregisterallevents` as follows:

- For a server with handle `h`, to unregister all events registered with a common event handling function `handler`, use:

```
h.unregisterevent('handler');
```

- To unregister individual events `eventN`, each registered with its own event handling function `handlerN`, use:

```
h.unregisterevent({'event1' 'handler1'; 'eventN' 'handlerN'});
```

- To unregister all events from the server regardless of which event handling function they are registered with, use:

```
h.unregisterallevents;
```

Responding to Events — Examples

The following examples show you how to respond to events from different COM objects:

Responding to Events from an ActiveX Control

This example describes how to handle events fired by an ActiveX control. It uses a control called `mwsamp2` that ships with MATLAB.

Tasks described in this section are:

Creating Event Handler Routines. You can view the event handler M-files for the `mwsamp2` control in the section "Sample Event Handlers" on page 9-66. Create the event handler files `myclick.m`, `my2click.m`, and `mymoused.m` and save them on your path, for example, `c:\work`.

Creating a Control and Registering Events. The `actxcontrol` function not only creates the control object, but you can use it to register specific events, as well. The code shown here registers two events (`Click` and `MouseDown`) and two respective handler routines (`myclick` and `mymoused`) with the `mwsamp2` control:

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

If, at some later time, you want to register additional events, use the `registerevent` function. For example:

```
obj.registerevent({'DblClick' 'my2click'});
```

Unregister the `DblClick` event before continuing with the example:

```
obj.unregisterevent({'DblClick' 'my2click'});
```

Listing Control Events. At this point, only the `Click` and `MouseDown` events should be registered. To list all events, whether registered or not, type:

```
objEvents = obj.events
```

MATLAB displays:

```
objEvents =
          Click: 'void Click()'
       DblClick: 'void DblClick()'
      MouseDown: 'void MouseDown(int16 Button, int16 Shift,
                   Variant x, Variant y)'
     Event_Args: [1x101 char]
```

This function returns a structure array, where each field of the structure is the name of an event handler and the value of that field contains the signature for the handler routine. For example:

```
objEvents.Event_Args
```

MATLAB displays:

```
ans =
    void Event_Args(int16 typeshort, int32 typelong,
        double typedouble, string typestring, bool typebool)
```

To list only the currently registered events, use the `eventlisteners` function:

```
obj.eventlisteners
```

MATLAB displays:

```
ans =
```

```
'click'        'myclick'
'mousedown'    'mymoused'
```

This function returns a cell array, with each row representing a registered event and the name of its event handler.

Responding to Control Events. When MATLAB creates the mwsamp2 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 see a report in the MATLAB Command Window of the X and Y position of the mouse.

Each time you press the mouse button, the MouseDown event fires, calling the mymoused function. This function prints the position values for that event to the Command Window. For example:

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

The Click event displays the message:

```
Single click function
```

Double-clicking the mouse does nothing different, since the DblClick event is not registered.

Unregistering Control Events. When you unregister an event, the client discontinues listening for occurrences of that event. When the event fires, the client does not respond. If you unregister the MouseDown event, MATLAB no longer reports the X and Y positions. Type:

```
obj.unregisterevent({'MouseDown' 'mymoused'});
```

When you click in the figure window, MATLAB displays:

```
Single click function
```

Now, register the DblClick event, using the my2click event handler:

```
obj.registerevent({'DblClick', 'my2click'});
```

If you call `eventlisteners` again:

```
obj.eventlisteners
```

MATLAB displays:

```
ans =
    'click'        'myclick'
    'dblclick'     'my2click'
```

When you double-click the mouse button, MATLAB displays:

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

An easy way to unregister all events for a control is to use the `unregisterallevents` function.

```
obj.unregisterallevents
obj.eventlisteners
```

When there are no events registered, `eventlisteners` returns an empty cell array:

```
ans =
    {}
```

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

Using a Common Event Handling Routine. If you have events that are registered with a common event handling routine, such as `sampev.m` used in the following example, you can use `unregisterevent` to unregister all of these events in one operation. This example first registers all events from the server with a common handling routine `sampev.m`. MATLAB now handles any type of event from this server by executing `sampev`:

```
obj.registerevent('sampev');
```

Verify the registration by listing all event listeners for that server:

```
    obj.eventlisteners
```

MATLAB displays:

```
ans =
    'click'        'sampev'
    'dblclick'     'sampev'
    'mousedown'    'sampev'
```

Now unregister all events for the server that use the sampev event handling routine:

```
    obj.unregisterevent('sampev');
    obj.eventlisteners
```

MATLAB displays:

```
ans =
    {}
```

Close the figure window.

Responding to Events from an Automation Server

This example shows how to handle events fired by an Automation server. It creates a server running the Microsoft Internet Explorer program, registers a common event handler for all events, and then has you fire events by browsing to Web sites.

Tasks described in this section are:

- "Creating an Event Handler" on page 9-61

- "Creating a Server" on page 9-61

- "Listing Server Events" on page 9-61

- "Registering Server Events" on page 9-62

- "Responding to Server Events" on page 9-62

- "Unregistering Server Events" on page 9-62

- "Closing the Application" on page 9-62

Creating an Event Handler. Register all events with the same handler routine, serverevents. Create the file serverevents.m, inserting the following code. Make sure the file is in your current directory.

```
function serverevents(varargin)

% Display incoming event name
eventname = varargin{end}

% Display incoming event args
eventargs = varargin{end-1}
```

Creating a Server. Next, at the MATLAB command prompt, type the following commands:

```
% Create a server running Internet Explorer.
browser = actxserver('internetexplorer.application');
% Make the server application visible.
browser.set('Visible', 1);
```

Listing Server Events. Use the events function to list all events the server can respond to, and eventlisteners to list the registered events:

```
browser.events
```

MATLAB displays event information similar to:

```
        :
StatusTextChange = void StatusTextChange(string Text)
ProgressChange = void ProgressChange(int32 Progress,int32 ProgressMax)
CommandStateChange = void CommandStateChange(int32 Command,bool Enable)
        :
```

List the registered events:

```
browser.eventlisteners
```

No events are registered at this time, so MATLAB displays:

```
ans =
     {}
```

Registering Server Events. Now use your event handler serverevents.

```
browser.registerevent('serverevents');
browser.eventlisteners
```

MATLAB displays:

```
ans =
            :                          :
    'statustextchange'      'serverevents'
    'progresschange'        'serverevents'
    'commandstatechange'    'serverevents'
            :                          :
```

Responding to Server Events. At this point, all events have been registered. If any event fires, the common handler routine defined in serverevents.m executes to handle that event. Use the Internet Explorer software to browse your favorite Web site, or enter the following command in the MATLAB Command Window:

```
browser.Navigate2('http://www.mathworks.com');
```

You should see a long series of events displayed in the Command Window.

Unregistering Server Events. Use the unregisterevent function to unregister the progresschange and commandstatechange events:

```
browser.unregisterevent({'progresschange', 'serverevents'; ...
    'commandstatechange', 'serverevents'});
```

To unregister all events for an object, use unregisterallevents. The following commands unregister all the events that had been registered, and then registers a single event:

```
browser.unregisterallevents;
browser.registerevent({'TitleChange', 'serverevents'});
```

If you now use the Web browser, MATLAB only responds to the TitleChange event.

Closing the Application. You should close a server application when you no longer intend to use it. To unregister all events and close the application, type:

```
browser.unregisterallevents;
browser.Quit;
browser.delete;
```

Responding to Interface Events from an Automation Server

This example, demonstrating how to handle a COM interface event, shows how to set up an event in an Microsoft Excel workbook object and how to handle its BeforeClose event.

To create the event handler OnBeforeCloseWorkbook, create the file OnBeforeCloseWorkbook.m, inserting the following code. Make sure the file is in your current directory:

```
% Event handler for Excel workbook BeforeClose event
function OnBeforeCloseWorkbook(varargin)
    disp('BeforeClose event occured');
```

When you run the following commands:

```
% Create Excel automation server instance
xl = actxserver('Excel.Application');
% Make it visible
xl.Visible = 1;

% Get collection of workbooks and add a new workbook
hWbks = xl.Workbooks;
hWorkbook = hWbks.Add;

% Register OnClose event
hWorkbook.registerevent({'BeforeClose' @OnBeforeCloseWorkbook});

% Close the workbook. This fires the Close event
% and calls the OnClose handler
hWorkbook.Close
```

MATLAB displays:

```
BeforeClose event occured
```

Writing Event Handlers

This section covers the following topics on writing handler routines to respond to events fired from a COM object:

- "Overview of Event Handling" on page 9-64
- "Arguments Passed to Event Handlers" on page 9-65
- "Event Structure" on page 9-66

Overview of Event Handling

An event is fired when a control or server wants to notify its client that something of interest has occurred. For example, many controls trigger an event when the user clicks somewhere in the interface window of a control. In MATLAB, you can create and register your own M-file functions to 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.

For controls, you can register handler functions either at the time you create the control (using `actxcontrol`), or at any time afterwards (using `registerevent`).

Both `actxcontrol` and `registerevent` use an event handler argument. 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. Strings used in the event handler argument are not case sensitive.

For servers, you must use `registerevent` to register those events you want the client to listen to. For example, to register the `Click` and `DblClick` events, use:

```
h.registerevent({'click' 'myclick'; 'dblclick' 'my2click'});
```

Use `events` to list all the events a COM object recognizes. For example, to list all events for the `mwsamp2` control, use:

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

```
h.events
   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, the MATLAB software passes information from the event to its handler function, as shown in this table.

Arguments Passed by MATLAB Functions

Arg. No.	Contents	Format
1	Object name	MATLAB COM class
2	Event ID	double
3	Start of Event Argument List	As passed by the control
end-2	End of Event Argument List (Argument 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 fields shown in the following table.

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.	Event Arg N	As passed by the control

For example, when the MouseDown event of the mwsamp2 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{5})
disp('The Y position is: ')
double(varargin{6})
```

Alternatively, you can use the same event handler for all the events you want to monitor using the cell array pairs. Response time is 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 (strcmp(varargin{end-1}.Type, 'Click'))
   disp ('Single Click Event Fired')
elseif (strcmp(varargin{end-1}.Type, 'DblClick'))
   disp ('Double Click Event Fired')
elseif (strcmp(varargin{end-1}.Type, 'MouseDown'))
   disp ('Mousedown Event Fired')
end
```

Writing Event Handlers Using M-File Subfunctions

Instead of having to maintain a separate M-file for every event handler routine you write, you can consolidate some or all of these routines into a single M-file using M-file subfunctions.

This example shows three event handler routines, (`myclick`, `my2click`, and `mymoused`) implemented as subfunctions in the file `mycallbacks.m`. The call to `str2func` converts the input string to a function handle:

```
function a = mycallbacks(str)
a = str2func(str);

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{5})
disp('The Y position is: ')
double(varargin{6})
```

To register one of these events, call `mycallbacks`, passing the name of the event handler:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
    gcf, 'sampev')
h.registerevent({'Click', mycallbacks('myclick')});
```

Getting Interfaces to the Object

In this section...
"IUnknown and IDispatch Interfaces" on page 9-69
"Custom Interfaces" on page 9-70

IUnknown and IDispatch Interfaces

When you invoke the `actxserver` or `actxcontrol` functions, the MATLAB software creates the server and returns a handle to the server interface as a means of accessing its properties and methods. The software uses the following process to determine which handle to return:

1 First get a handle to the IUnknown interface from the component. All COM components are required to implement this interface.

2 Attempt to get the IDispatch interface. If IDispatch is implemented, return a handle to this interface. If IDispatch is not implemented, return the handle to IUnknown.

Additional Interfaces

Components often provide additional interfaces, based on IDispatch, that are implemented as properties. Like any other property, you obtain these interfaces using the MATLAB `get` function.

For example, a Microsoft Excel component contains numerous interfaces. To list these interfaces, along with Excel properties, type:

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

MATLAB displays information similar to:

```
    Application: [1x1 Interface.Microsoft_Excel_9.0_
Object_Library._Application]
        Creator: 'xlCreatorCode'
         Parent: [1x1 Interface.Microsoft_Excel_9.0_
Object_Library._Application]
```

```
          ActiveCell: []
         ActiveChart: [1x50 char]
                      .
                      .
```

To see if `Workbooks` is an interface, type:

```
w = h.Workbooks
```

MATLAB displays:

```
w =
    Interface.Microsoft_Excel_9.0_Object_Library.Workbooks
```

The information displayed depends on the version of the Excel software you have on your system.

Custom Interfaces

The following client/server configurations support interface:

- "MATLAB Client and In-Process Server" on page 8-32
- "MATLAB Client and Out-of-Process Server" on page 8-33

Note The MATLAB COM Interface does not support invoking functions with optional parameters.

Once you have created the server, you can query the server component to see if any custom interfaces are implemented using the `interfaces` function. `interfaces` returns the names in a cell array of strings.

For example, if you have a component with the ProgID `mytestenv.calculator`, you can see its custom interfaces using the commands:

```
h = actxserver('mytestenv.calculator');
customlist = h.interfaces
```

MATLAB displays the interfaces, which might be:

```
customlist =
    ICalc1
    ICalc2
    ICalc3
```

To get the handle to a particular interface, use the `invoke` function

```
c1 = h.invoke('ICalc1')
c1 =
    Interface.Calc_1.0_Type_Library.ICalc_Interface
```

Use this handle `c1` to access the properties and methods of the object through this custom interface `ICalc1`.

For example, to list the properties, use:

```
c1.get
     background: 'Blue'
         height: 10
          width: 0
```

To list the methods, use:

```
c1.invoke
    Add = double Add(handle, double, double)
    Divide = double Divide(handle, double, double)
    Multiply = double Multiply(handle, double, double)
    Subtract = double Subtract(handle, double, double)
```

To add and multiply numbers using the `Add` and `Multiply` methods of the object, use:

```
sum = c1.Add(4, 7)
sum =
    11

prod = c1.Multiply(4, 7)
prod =
    28
```

Saving Your Work

In this section...
"Functions for Saving and Restoring COM Objects" on page 9-72
"Releasing COM Interfaces and Objects" on page 9-73

Functions for Saving and Restoring COM Objects

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

Function	**Description**
load	Load and initialize a COM control object from a file
save	Write and serialize a COM control object to a file

Save, or *serialize*, the current state of a COM control to a file using the save function. *Serialization* is the process of saving an object onto a storage medium (such as a file or a memory buffer) or transmitting it across a network connection link in binary form.

The following example creates an mwsamp2 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);
h.save('mwsample')
```

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

```
h.Label = 'Circle';
h.Radius = 50;
h.Redraw;
```

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

```
h.load('mwsample');
```

To verify the results, type:

```
h.get
```

MATLAB displays:

```
ans =
      Label: 'Label'
     Radius: 20
```

Note MATLAB supports the COM `save` and `load` functions for controls only.

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 interface

When you no longer need an interface, use the `release` function to release the interface and reclaim the memory used by it. When you no longer need a control or server, use the `delete` function to delete it. Alternatively, you can use the `delete` function to both release all interfaces for the object and delete the server or control.

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 version 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.

When you delete or close a figure window containing a control, MATLAB automatically releases all interfaces for the control. MATLAB also automatically releases all handles for an Automation server when you exit the program.

Handling COM Data in MATLAB Software

Passing Data to a COM Object

When you use a COM object in a MATLAB command, the MATLAB types you pass in the call are converted to types native to the COM object. MATLAB performs this conversion on each argument that is passed. This section describes the conversion.

MATLAB arguments are converted by MATLAB into types that best represent the data to the COM object. The following table shows all of the MATLAB base types for passed arguments and the COM types defined for input arguments. Each row shows a MATLAB type followed by the possible COM argument matches.

MATLAB Argument	**Closest Type**	**Allowed Types**
handle	VT_DISPATCH VT_UNKNOWN	VT_DISPATCH VT_UNKNOWN

MATLAB Argument	Closest Type	Allowed Types
string	VT_BSTR	VT_LPWSTR VT_LPSTR VT_BSTR VT_FILETIME VT_ERROR VT_DECIMAL VT_CLSID VT_DATE
int16	VT_I2	VT_UINT VT_I2 VT_UI2
int32	VT_I4	VT_I4 VT_UI4 VT_INT
single	VT_R4	VT_R4
double	VT_R8	VT_R8 VT_CY (currency)
bool	VT_BOOL	VT_BOOL
char	VT_I1	VT_I1 VT_UI1

Variant Data

variant is any data type except a structure or a sparse array. (Refer to the Data Type Summary table in the MATLAB Programming Fundamentals documentation.)

When used as an input argument, MATLAB treats variant and variant(pointer) the same way.

MATLAB Argument	Closest Type	Allowed Types
variant	VT_VARIANT	VT_VARIANT VT_USERDEFINED VT_ARRAY
variant(pointer)	VT_VARIANT	VT_VARIANT | VT_BYREF

SAFEARRAY Data

When a COM method identifies a SAFEARRAY or SAFEARRAY(pointer), the MATLAB equivalent is a matrix.

MATLAB Argument	Closest Type	Allowed Types
SAFEARRAY	VT_SAFEARRAY	VT_SAFEARRAY
SAFEARRAY(pointer)	VT_SAFEARRAY	VT_SAFEARRAY | VT_BYREF

Handling Data from a COM Object

Data returned from a COM object is often incompatible with MATLAB types. When this occurs, MATLAB converts the returned value to a data type native to the MATLAB language. This section describes the conversion performed on the various types that can be returned from COM objects.

The following table shows how MATLAB converts data from a COM object into MATLAB variables.

COM Return Type	MATLAB Representation
VT_DISPATCH VT_UNKNOWN	handle
VT_LPWSTR VT_LPSTR VT_BSTR VT_FILETIME VT_ERROR VT_DECIMAL VT_CLSID VT_DATE	string

COM Return Type	MATLAB Representation

`VT_INT`
`VT_UINT`
`VT_I2`
`VT_UI2`
`VT_I4`
`VT_UI4`
`VT_R4`
`VT_R8`
`VT_CY` (currency)
	`double`

`VT_BOOL`	`logical`
`VT_I1` `VT_UI1`	`char`

`VT_VARIANT`
`VT_USERDEFINED`
`VT_ARRAY`
	`variant`

`VT_VARIANT	VT_BYREF`	`variant`(pointer)
`VT_SAFEARRAY`	`SAFEARRAY`
`VT_SAFEARRAY	VT_BYREF`	`SAFEARRAY`(pointer)

Unsupported Types

MATLAB does not support the following COM interface types:

- `VT_I8`

- `VT_UI8`

- Structure

- Sparse array

- Unsigned integer

- Multidimensional `SAFEARRAY`s

- Write-only properties

- Enumerated types

Passing MATLAB Data to ActiveX Objects

The tables also show the mapping of MATLAB types to COM types that you must use to pass data from MATLAB to an Microsoft ActiveX object. Note that all other types result in the following warning:

```
"ActiveX - invalid argument type or value".
```

Passing MATLAB SAFEARRAY to COM Object

The SAFEARRAY data type is a standard way to pass arrays between COM objects. This section explains how MATLAB passes SAFEARRAY data to a COM object.

- "Default Behavior in MATLAB Software" on page 9-78

- "Examples" on page 9-78

- "How to Pass a Single-Dimension SAFEARRAY" on page 9-80

- "Passing SAFEARRAY By Reference" on page 9-80

Default Behavior in MATLAB Software

MATLAB represents an m-by-n matrix as a two-dimensional SAFEARRAY, where the first dimension has m elements and the second dimension has n elements. MATLAB passes the SAFEARRAY by value.

Examples

The following examples use a COM object that expects a SAFEARRAY input parameter.

When MATLAB passes a 1-by-3 array :

```
B = [2 3 4]
B =
     2     3     4
```

the object reads:

```
No. of dimensions: 2
Dim: 1,   No. of elements: 1
```

```
    Dim: 2,   No. of elements: 3
        Elements:
        2.0
        3.0
        4.0
```

When MATLAB passes a 3-by-1 array:

```
C = [1;2;3]
C =
     1
     2
     3
```

the object reads:

```
No. of dimensions: 2
Dim: 1,   No. of elements: 3
Dim: 2,   No. of elements: 1
Elements:
    1.0
    2.0
    3.0
```

When MATLAB passes a 2-by-4 array:

```
D = [2 3 4 5;5 6 7 8]

D =
     2     3     4     5
     5     6     7     8
```

the object reads:

```
No. of dimensions: 2
Dim: 1,   No. of elements: 2
Dim: 2,   No. of elements: 4
Elements:
    2.0
    3.0
    4.0
```

```
5.0
5.0
6.0
7.0
8.0
```

How to Pass a Single-Dimension SAFEARRAY

For information about passing arguments as one-dimensional arrays to a COM object, see the Technical Support solution 1-SKYP9 at http://www.mathworks.com/support/solutions/data/1-SKYP9.html.

Passing SAFEARRAY By Reference

For information about passing arguments by reference to a COM object, see the Technical Support solution 1-SKYPY at http://www.mathworks.com/support/solutions/data/1-SKYPY.html.

Reading SAFEARRAY from a COM Object in MATLAB Applications

This section explains how MATLAB reads SAFEARRAY data from a COM object.

MATLAB reads a one-dimensional SAFEARRAY with n elements from a COM object as a 1-by-n matrix. For example, using methods from the MATLAB sample control mwsamp, type:

```
h=actxcontrol('mwsamp.mwsampctrl.1')
a = h.GetI4Vector
```

MATLAB displays:

```
a =
          1          2          3
```

MATLAB reads a two-dimensional SAFEARRAY with n elements as a 2-by-n matrix. For example:

```
a = h.GetR8Array
```

MATLAB displays:

```
a =
     1     2     3
     4     5     6
```

MATLAB reads a three-dimensional `SAFEARRAY` with 2 elements as a 2-by-2-by-2 cell array. For example:

```
a = h.GetBSTRArray
```

MATLAB displays:

```
a(:,:,1) =

    '1 1 1'    '1 2 1'
    '2 1 1'    '2 2 1'

a(:,:,2) =

    '1 1 2'    '1 2 2'
    '2 1 2'    '2 2 2'
```

Displaying MATLAB Syntax for COM Objects

To determine which MATLAB types to use when passing arguments to COM objects, use the `invoke` or `methodsview` functions. These functions list all of the methods found in an object, along with a specification of the types required for each argument.

In the following example, a server called `MyApp` has a method `TestMeth1` with the following syntax:

```
HRESULT TestMeth1 ([out, retval] double* dret);
```

This method has no input argument, and it returns a variable of type `double`. To display the MATLAB syntax for calling the method, type:

```
h = actxserver('MyApp');
h.invoke
```

MATLAB displays:

```
ans =
    TestMeth1 = double TestMeth1 (handle)
```

The signature of `TestMeth1` is:

```
double TestMeth1(handle)
```

MATLAB requires you to use an object handle as an input argument for every method, in addition to any input arguments required by the method itself.

Using the variable `var`, which is of type `double`, type:

```
var = h.TestMeth1;
```

or:

```
var = TestMeth1(h);
```

While the following syntax is correct, its use is discouraged:

```
var = invoke(h,'TestMeth1');
```

Now consider the server called `MyApp1` with the following methods:

```
HRESULT TestMeth1 ([out, retval] double* dret);
HRESULT TestMeth2 ([in] double* d, [out, retval] double* dret);
HRESULT TestMeth3 ([out] BSTR* sout,
                   [in, out] double* dinout,
                   [in, out] BSTR* sinout,
                   [in] short sh,
                   [out] long* ln,
                   [in, out] float* b1,
                   [out, retval] double* dret);
```

Type the commands:

```
h = actxserver('MyApp1');
h.invoke
```

MATLAB displays the list of methods:

```
ans =
```

```
TestMeth1 = double TestMeth1 (handle)
TestMeth2 = double TestMeth1 (handle, double)
TestMeth3 = [double, string, double, string, int32, single] ...
            TestMeth3(handle, double, string, int16, single)
```

`TestMeth2` requires an input argument d of type `double`, as well as returning a variable `dret` of type `double`. Some examples of calling `TestMeth2` are:

```
var = h.TestMeth2(5);
```

or:

```
var = TestMeth2(h, 5);
```

`TestMeth3` requires multiple input arguments, as indicated within the parentheses on the right side of the equals sign, and returns multiple output arguments, as indicated within the brackets on the left side of the equals sign.

```
[double, string, double, string, int32, single]  %output arguments
TestMeth3(handle, double, string, int16, single) %input arguments
```

The first input argument is the required `handle`, followed by four input arguments.

```
TestMeth3(handle, in1, in2, in3, in4)
```

The first output argument is the return value `retval`, followed by five output arguments.

```
[retval, out1, out2, out3, out4, out5]
```

This is how the arguments map into a MATLAB command:

```
[dret, sout, dinout, sinout, ln, b1] = TestMeth3(handle, ...
                                      dinout, sinout, sh, b1)
```

where `dret` is `double`, `sout` is `string`, `dinout` is `double` and is both an input and an output argument, `sinout` is `string` (input and output argument), `ln` is `int32`, `b1` is `single` (input and output argument), `handle` is the handle to the object, and `sh` is `int16`.

Examples of MATLAB Software as an Automation Client

In this section...
"MATLAB Sample Control" on page 9-84
"Using a MATLAB Application as an Automation Client" on page 9-84
"Connecting to an Existing Excel Application" on page 9-86
"Running a Macro in an Excel Server Application" on page 9-87
"MATLAB COM Client Demo" on page 9-88

MATLAB Sample Control

MATLAB software ships with a simple example COM control that draws a circle on the screen, displays some text, and fires events when the user single- or double-clicks 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]);
```

You can find this control 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. For other examples using the mwsamp2 control, see "Writing Event Handlers" on page 9-64.

Using a MATLAB Application as an Automation Client

This example uses MATLAB software as an Automation client and the Microsoft Excel spreadsheet program 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.
eWorkbook = e.Workbooks.Add;
e.Visible = 1;

% Make the first sheet active.
eSheets = e.ActiveWorkbook.Sheets;

eSheet1 = eSheets.get('Item', 1);
eSheet1.Activate;

% Put a MATLAB array into Excel.
A = [1 2; 3 4];
eActivesheetRange = e.Activesheet.get('Range', 'A1:B2');
eActivesheetRange.Value = A;

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

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

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

% Avoid saving the workbook and being prompted to do so
eWorkbook.Saved = 1;
eWorkbook.Close;

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

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

Connecting to an Existing Excel Application

You can give MATLAB access to a file that is open by another application by creating a new COM server from the MATLAB client, and then opening the file through this server. This example shows how to do this for an Excel application that has a file weekly_log.xls open:

```
excelapp = actxserver('Excel.Application');
wkbk = excelapp.Workbooks;
wdata = wkbk.Open('d:\weatherlog\weekly_log.xls');
```

To see what methods are available, type:

```
wdata.methods
   Methods for class Interface.Microsoft_Excel_10.0_
      Object_Library._Workbook:

   AcceptAllChanges    LinkInfo      ReloadAs
   Activate            LinkSources   RemoveUser
      :                   :              :
      :                   :              :
```

Access data from the spreadsheet by selecting a particular sheet (called 'Week 12' in the example), selecting the range of values (the rectangular area defined by D1 and F6 here), and then reading from this range:

```
sheets = wdata.Sheets;
sheet12 = sheets.Item('Week 12');
range = sheet12.get('Range', 'D1', 'F6');
range.value

ans =
    'Temp.'       'Heat Index'    'Wind Chill'
    [78.4200]     [        32]    [        37]
    [69.7300]     [        27]    [        30]
    [77.6500]     [        17]    [        16]
    [74.2500]     [        -5]    [         0]
    [68.1900]     [        22]    [        35]

wkbk.Close;
excelapp.Quit;
```

Running a Macro in an Excel Server Application

In the following example, MATLAB runs the Microsoft Excel spreadsheet program in a COM server and invokes a macro that has been defined within the active Excel spreadsheet file. The macro, init_last, takes no input parameters and is called from the MATLAB client using the statement:

```
handle.ExecuteExcel4Macro('!macroname()');
```

Start the example by opening the spreadsheet file and recording a macro. The macro used here simply sets all items in the last column to zero. Save your changes to the spreadsheet.

Next, in MATLAB, create a COM server running an Excel application, and open the spreadsheet:

```
h = actxserver('Excel.Application');
wkbk = h.Workbooks;
file = wkbk.Open('d:\weatherlog\weekly.xls');
```

Open the sheet that you want to change, and retrieve the current values in the range of interest:

```
sheets = file.Sheets;
sheet12 = sheets.Item('Week 12');
range = sheet12.get('Range', 'D1', 'F5');
range.Value
ans =
    [     78]    [      32]    [      37]
    [     69]    [      27]    [      30]
    [     77]    [      17]    [      16]
    [     74]    [      -5]    [      -1]
    [     68]    [      22]    [      35]
```

Now execute the macro, and verify that the values have changed as expected:

```
h.ExecuteExcel4Macro('!init_last()');
range.Value
ans =
    [     78]    [      32]    [       0]
    [     69]    [      27]    [       0]
    [     77]    [      17]    [       0]
```

```
[      74]    [         -5]    [          0]
[      68]    [         22]    [          0]
```

MATLAB COM Client Demo

MATLAB includes a demo illustrating the use of the COM Client with MATLAB. To run the demo, click the **Demos** tab in the MATLAB Help browser. Click to expand the folder called External Interfaces and select Programming with COM.

Advanced Topics

In this section...
"Deploying ActiveX Controls Requiring Run-Time Licenses" on page 9-89
"Using Microsoft Forms 2.0 Controls" on page 9-90
"Using COM Collections" on page 9-91
"Using MATLAB Application as a DCOM Client" on page 9-92
"MATLAB COM Support Limitations" on page 9-92

Deploying ActiveX Controls Requiring Run-Time Licenses

When you deploy a Microsoft ActiveX control that requires a run-time license, you must include a license key, which the control reads at run-time. If the key matches the control's own version of the license key, an instance of the control is created. Use the following procedure to deploy a run-time-licensed control with a MATLAB application.

Create an M-File to Build the Control

First, create an M-file to build the control. This M-file must contain two elements:

- The pragma %#function actxlicense. This pragma causes the MATLAB Compiler to embed a function named actxlicense into the stand alone executable file you build.

- A call to actxcontrol to create the control.

Place this M-file in a directory outside of the MATLAB code tree.

Here is an example M-file.

```
function buildcontrol
%#function actxlicense
h=actxcontrol('MFCCONTROL2.MFCControl2Ctrl.1',[10 10 200 200]);
```

Build the Control and the License M-File

Change to the directory where you placed the M-file you created to build the control. Call the function you defined in the M-file. When it executes this function, MATLAB determines whether the control requires a run-time license. If it does, MATLAB creates another M-file, named `actxlicense.m`, in the current working directory. The `actxlicense` function defined in this file provides the license key to MATLAB at run-time.

Build the Executable

Next, call MATLAB Compiler to create the stand alone executable from the file you created to build the control. The executable contains both the function that builds the control and the `actxlicense` function.

```
mcc -m buildcontrol
```

Deploy the Files

Finally, distribute `buildcontrol.exe`, `buildcontrol.ctf`, and the control (`.ocx` or `.dll`).

Using Microsoft Forms 2.0 Controls

You may encounter problems when creating or using Microsoft Forms 2.0 controls in MATLAB. Forms 2.0 controls are designed for use with applications enabled by Microsoft Visual Basic for Applications (VBA). An example is Microsoft Office software.

To work around these problems, use the following replacement controls, or consult article 236458 in the Microsoft Knowledge Base for further information:

```
http://support.microsoft.com/default.aspx?kbid=236458
```

Affected Controls

You may see this behavior with any of the following Forms 2.0 controls:

- `Forms.TextBox.1`
- `Forms.CheckBox.1`

- `Forms.CommandButton.1`

- `Forms.Image.1`

- `Forms.OptionButton.1`

- `Forms.ScrollBar.1`

- `Forms.SpinButton.1`

- `Forms.TabStrip.1`

- `Forms.ToggleButton.1`

Replacement Controls

Microsoft recommends the following replacements:

Old	New
`Forms.TextBox.1`	`RICHTEXT.RichtextCtrl.1`
`Forms.CheckBox.1`	`vidtc3.Checkbox`
`Forms.CommandButton.1`	`MSComCtl2.FlatScrollBar.2`
`Forms.TabStrip.1`	`COMCTL.TabStrip.1`

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, you should release this interface 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 = hControl.Plots;
for i = 1:hCollection.Count
   hPlot = hCollection.invoke('Item', i);
   hPlot.Redraw;
   hPlot.release;
end;
hCollection.release;
```

Using MATLAB Application as a DCOM Client

Distributed Component Object Model (DCOM) is a protocol that allows clients to use remote COM objects over a network. Additionally, MATLAB can be used as a DCOM client with remote Automation servers if the operating system on which MATLAB is running is DCOM enabled.

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

MATLAB COM Support Limitations

Limitations of MATLAB COM support are:

- MATLAB only supports indexed collections.

- COM controls are not printed with figure windows.

- "Unsupported Types" on page 9-77

10

MATLAB COM Automation Server Support

Introduction

In this section...

What Is Automation?

Automation is a COM protocol that allows one application (the *controller* or *client*) to control objects exported by another application (the *server*). MATLAB software on Microsoft Windows operating systems supports COM Automation server capabilities. Any Windows program that can be configured as an Automation controller can control MATLAB. Some examples of applications that can be Automation controllers are Microsoft Excel, Microsoft Access, and Microsoft Project applications, and many Microsoft Visual Basic and Microsoft Visual C++ programs.

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

Creating the MATLAB Server

To create a server, you need a programmatic identifier (ProgID) to identify the server. The ProgID for MATLAB is matlab.application. For other a list of MATLAB ProgIDs, see "Programmatic Identifiers" on page 8-4.

How you create an Automation server depends on the controller you are using. Consult your controller's documentation for this information.

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

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

This command automatically creates the Automation server. You can also create the server manually. See "Creating the Server Manually" on page 10-13.

The following topics:

- "Using MATLAB Software as a Shared or Dedicated Server" on page 10-3
- "Accessing Your Server from the Startup Directory" on page 10-3
- "Get the Status of a MATLAB Automation Server" on page 10-4
- "Creating a MATLAB Automation Server from Visual Basic .NET Application" on page 10-4

Using MATLAB Software as a Shared or Dedicated Server

The MATLAB Automation server has two modes:

- Shared — One or more client applications connect to the same MATLAB server. All clients share the same server.
- Dedicated — Each client application creates its own dedicated MATLAB server.

If you use `matlab.application` as your ProgID, MATLAB creates a shared server. For information about creating shared and dedicated servers, see "Specifying a Shared or Dedicated Server" on page 10-14.

Accessing Your Server from the Startup Directory

The MATLAB Automation server starts up in the *matlabroot*\bin\win32 directory. If this is not the MATLAB client startup directory, the newly created server does not run the MATLAB startup file (`startup.m`) and does not have access to files in that directory.

To access files in the startup directory, do one of the following:

- Set the server's working directory to the startup directory (using the `cd` function) and add the startup directory to the server's MATLAB path (using the `addpath` function).
- Include the path name to the startup directory when referencing those files.

Get the Status of a MATLAB Automation Server

Use the `enableservice` function to determine the current state of a MATLAB Automation server. The function returns a logical value, where logical 1 (`true`) means MATLAB is an Automation server and logical 0 (`false`) means MATLAB is not an Automation server.

For example, if you type:

```
enableservice('AutomationServer')
```

and MATLAB displays:

```
ans =
     1
```

then MATLAB is currently an Automation server.

Creating a MATLAB Automation Server from Visual Basic .NET Application

If you use a Visual Basic client application to access a MATLAB Automation server, you have two options for creating the server:

- "Accessing Methods from the Visual Basic Object Browser" on page 10-4
- "Using CreateObject" on page 10-5

Accessing Methods from the Visual Basic Object Browser. You can use the Object Browser of your Visual Basic client application to see what methods are available from a MATLAB Automation server. To do this you need to reference the MATLAB *type library* in your Visual Basic project.

To set up your Visual Basic project:

1 Select the **Project** menu.

2 Select **Reference** from the subsequent menu.

3 Check the box next to the **MATLAB Application Type Library**.

4 Click **OK**.

In your Visual Basic code, use the `New` method to create the server:

```
Matlab = New MLApp.MLApp
```

View MATLAB Automation methods from the Visual Basic Object Browser under the Library called `MLAPP`.

Using CreateObject. To use the Visual Basic `CreateObject` method, type:

```
MatLab = CreateObject("Matlab.Application")
```

Connecting to an Existing MATLAB Server

It is not always necessary to create a new instance of a MATLAB server whenever your application needs some task done in MATLAB. Clients can connect to an existing MATLAB Automation server using the `actxGetRunningServer` function or by using a command similar to the Visual Basic .NET `GetObject` command.

Using Visual Basic .NET Code

The Visual Basic .NET command shown here returns a handle `h` to the MATLAB server application:

```
h = GetObject(, "matlab.application")
```

Note It is important to use the syntax shown above to connect to an existing MATLAB Automation server. Omit the first argument, and make sure the second argument is as shown.

The following Visual Basic .NET example connects to an existing MATLAB server, then executes a plot command in the server. If you do not already have a MATLAB server running, create one following the instructions in "Creating a MATLAB Automation Server from Visual Basic .NET Application" on page 10-4.

```
Dim h As Object
h = GetObject(, "matlab.application")

' Handle h should be valid now.
```

```
' Test it by calling Execute.
h.Execute ("plot([0 18], [7 23])")
```

MATLAB Automation Server Functions and Properties

In this section...

Introduction

MATLAB functions and properties enable an Automation controller to manipulate data in the MATLAB workspace. MATLAB can be both a controller and a server. The examples in this section use a MATLAB M-file as the client application. For information about how to access a MATLAB server from other applications, see "Examples of a MATLAB Automation Server" on page 10-16.

This section explains how to call functions in the MATLAB Automation server and how to use properties that affect the server. These are shown in the following tables and are described in individual function reference pages.

For a complete list of these functions, see "Component Object Model and ActiveX" in the MATLAB Function Reference documentation.

Executing Commands in the MATLAB Server

The client program can execute commands in the MATLAB server using these functions.

Function	Description
Execute	Execute MATLAB command in server
Feval	Evaluate MATLAB command in server

Using Execute

Use the `Execute` function when you want the MATLAB server to execute a command that can be expressed in a single string. For example:

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

h.PutWorkspaceData('A', 'base', rand(6))
h.Execute('A(4:6,:) = [];'); % remove rows 4-6
B = h.GetWorkspaceData('A', 'base')
```

MATLAB displays:

```
B =
    0.6208    0.2344    0.6273    0.3716    0.7764    0.7036
    0.7313    0.5488    0.6991    0.4253    0.4893    0.4850
    0.1939    0.9316    0.3972    0.5947    0.1859    0.1146
```

Using Feval

Use the `Feval` function when you want the server to execute commands that you cannot express in a single string. The following example uses variables defined in the client, `rows` and `cols`, to modify the server.

This is a continuation of the previous example:

```
rows = 6;    cols = 3;
h.Feval('reshape', 0, 'A=', rows, cols);
```

MATLAB interprets `A` in the expression `'A='` as a server variable name.

The `reshape` function in the previous statement does not make an assignment to the server variable A; it is equivalent to the following MATLAB statement:

```
reshape(A,6,3)
```

which returns a result, but does not assign the new array. If you get the variable A from the server, it is unchanged:

```
B = h.GetWorkspaceData('A', 'base')
```

MATLAB displays:

```
B =
    0.6208    0.2344    0.6273    0.3716    0.7764    0.7036
    0.7313    0.5488    0.6991    0.4253    0.4893    0.4850
    0.1939    0.9316    0.3972    0.5947    0.1859    0.1146
```

Use the `Feval` function return value to get the result of this type of operation. For example, the following statement reshapes the server-side array A and returns the result of this operation in the client-side variable a:

```
a = h.Feval('reshape', 1, 'A=', rows, cols);
```

The `Feval` function returns a cell array. To view the contents, type:

```
a{:}
```

MATLAB displays:

```
ans =
    0.6208    0.6273    0.7764
    0.7313    0.6991    0.4893
    0.1939    0.3972    0.1859
    0.2344    0.3716    0.7036
    0.5488    0.4253    0.4850
    0.9316    0.5947    0.1146
```

Exchanging Data with the Server

MATLAB provides functions to read and write data to any workspace of a MATLAB server. In these commands, pass the name of the variable to read or write, and the name of the workspace holding that data.

Function	Description
GetCharArray	Get character array from server
GetFullMatrix	Get matrix from server
GetWorkspaceData	Get any type of data from server
PutCharArray	Store character array in server

Function	Description
PutFullMatrix	Store matrix in server
PutWorkspaceData	Store any type of data in server

The Get/PutCharArray functions read and write string values to the MATLAB server.

The Get/PutFullMatrix functions pass data as a SAFEARRAY data type. You can use these functions with any client that supports the SAFEARRAY type. This includes MATLAB and Visual Basic clients.

The Get/PutWorkspaceData functions pass data as a variant data type. Use these functions with any client that supports the variant type. These functions are especially useful for VBScript clients because VBScript does not support the SAFEARRAY data type.

In this example, write a string to variable str in the base workspace of the MATLAB server and read it back to the client:

```
h = actxserver('matlab.application');
h.PutCharArray('str', 'base', ...
   'He jests at scars that never felt a wound.');

S = h.GetCharArray('str', 'base')
S =
  He jests at scars that never felt a wound.
```

Controlling the Server Window

These functions enable you to make the server window visible or to minimize it.

Function	Description
MaximizeCommandWindow	Display server window on Windows desktop
MinimizeCommandWindow	Minimize size of server window

In this example, create a COM server running MATLAB and minimize it:

```
h = actxserver('matlab.application');
h.MinimizeCommandWindow;
```

Terminating the Server Process

When you are finished with the MATLAB server, use these functions to quit the MATLAB session and terminate the server process.

Function	Description
Quit	Quit the MATLAB session
delete	Terminate MATLAB server process

To quit MATLAB, type:

```
h.Quit;
```

To terminate the server process, type:

```
h.delete;
```

Client-Specific Information

This section provides information specific to MATLAB and Visual Basic .NET clients only.

For MATLAB Clients

To see a summary of all functions along with the required syntax, use the invoke function as follows:

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

For Visual Basic .NET Clients

Data types for the arguments and return values of the server functions are expressed as Automation data types, which are language-independent types defined by the Automation protocol.

For example, BSTR is a wide-character string type defined as an Automation type, and is the same data format used by the Visual Basic language to store

strings. Any COM-compliant controller should support these data types, although the details of how you declare and manipulate these are controller specific.

Using the Visible Property

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

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

```
h = actxserver('matlab.application');
h.Visible
ans =
     1
```

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

```
h.Visible = 0;
h.Visible
ans =
     0
```

Additional Automation Server Information

In this section...

Creating the Server Manually

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

To manually create a MATLAB server, use the /Automation switch in the MATLAB startup command. You can do this from the DOS command line by typing:

```
matlab /Automation
```

Alternatively, you can add this switch every time you run MATLAB, as follows:

1 Right-click the MATLAB shortcut icon

and select **Properties** from the context menu. The Properties dialog box for matlab.exe opens to the **Shortcut** tab.

2 In the **Target** field, add /Automation to the end of the target path for matlab.exe. Be sure to include a space between the file name and the symbol /. For example:

```
"C:\Program Files\MATLAB\R2006a\bin\win32\MATLAB.exe /Automation"
```

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

Specifying a Shared or Dedicated Server

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

Starting a Shared Server

The ProgID, `matlab.application`, specifies the default mode, which is shared. You can also use the version-specific ProgID, `matlab.application.N.M`, where `N` is the major version and `M` is the minor version of your MATLAB. For example, use `N = 7` and `M = 4` for MATLAB version 7.4.

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

Starting a Dedicated Server

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

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

Using Date Data Type

When you need to pass a VT_DATE type input to a Visual Basic program or an ActiveX control method, you can use the MATLAB class COM.date. For example:

```
d = COM.date(2005,12,21,15,30,05);
get(d)
     Value: 7.3267e+005
    String: '12/21/2005 3:30:05 PM'
```

You can use now to set the Value property to a date number:

```
d.Value = now;
```

Using MATLAB Application as a DCOM Server

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

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

Examples of a MATLAB Automation Server

Example — Running an M-File from Visual Basic .NET Program

This example calls a user-defined M-file function named solve_bvp from a Microsoft Visual Basic client application through a COM interface. It also plots a graph in a new MATLAB window and performs a simple computation:

```
Dim MatLab As Object
Dim Result As String
Dim MReal(1, 3) As Double
Dim MImag(1, 3) As Double

MatLab = CreateObject("Matlab.Application")

'Calling m-file from VB
'Assuming solve_bvp exists at specified location
Result = MatLab.Execute("cd d:\matlab\work\bvp")
Result = MatLab.Execute("solve_bvp")

'Executing other MATLAB commands
Result = MatLab.Execute("surf(peaks)")
Result = MatLab.Execute("a = [1 2 3 4; 5 6 7 8]")
Result = MatLab.Execute("b = a + a ")
'Bring matrix b into VB program
MatLab.GetFullMatrix("b", "base", MReal, MImag)
```

Example — Viewing Methods from a Visual Basic .NET Client

You can find out what methods are available from a MATLAB Automation server using the Object Browser of your Microsoft Visual Basic client application. To do this, follow this procedure in the client application to reference the MATLAB Application Type Library:

1 Select the **Project** menu.

2 Select **Reference** from the subsequent menu.

3 Check the box next to the **MATLAB Application Type Library**.

4 Click **OK**.

This enables you to view MATLAB Automation methods from the Visual Basic Object Browser under the Library called MLAPP. You can also see a list of MATLAB Automation methods when you use the term Matlab followed by a period. For example:

```
Dim Matlab As MLApp.MLApp
Private Sub View_Methods()
Matlab = New MLApp.MLApp
'The next line shows a list of MATLAB Automation methods
Matlab.
End Sub
```

Example — Calling MATLAB Software from a Web Application

This example shows you how to create a Web page that uses a MATLAB application as an Automation server. For another example using ASP.NET, see Technical Support solution 1–3JJZWN at http://www.mathworks.com/support/solutions/data/1-3JJZWN.html.

You can invoke MATLAB as an Automation server from any language that supports COM, so for Web applications, you can use VBScript and JavaScript. While this example is simple, it illustrates techniques for passing commands to MATLAB and writing data to and retrieving data from the MATLAB

workspace. See "Exchanging Data with the Server" on page 10-9 for related functions.

VBScript and HTML forms are combined in this example to create an interface that enables the user to select a MATLAB plot type from a pull-down menu, click a button, and create the plot in a MATLAB figure window. To accomplish this, the HTML file contains code that:

- Starts MATLAB as an Automation server via a VBScript.
- When users click a button on the HTML page, a VBScript executes that:

  1 Determines the type of plot selected.

  2 Forms a command string to create the type of plot selected.

  3 Forms a string describing the type of plot selected, which passes to the MATLAB base workspace in a variable.

  4 Executes the MATLAB command.

  5 Retrieves the descriptive string from the MATLAB workspace.

  6 Updates the text box on the HTML page.

Here is the HTML used to create this example:

```
<HTML>
<HEAD>
<TITLE>Example of calling MATLAB from VBScript</TITLE>
</HEAD>
<BODY>
<FONT FACE = "Arial, Helvetica, Geneva" SIZE = "+1" COLOR = "maroon">
Example of calling MATLAB from VBScript
</FONT>
<FONT FACE = "Arial, Helvetica, Geneva" SIZE = "-1">

<!-- %%%%%%%%%%%%%%%%%%% BEGIN SCRIPT %%%%%%%%%%%%%%%%%%% -->
<SCRIPT LANGUAGE="VBScript">
<!-- Invoke MATLAB as a COM Automation server upon loading page
' Initialize global variables
Dim MatLab 'COM Automation server variable
Dim MLcmd 'string to send to MATLAB for execution
' Invoke COM Automation server
```

```
Set MatLab = CreateObject("Matlab.Application")
' End initialization script -->
</SCRIPT>

<!-- %%%%%%%%%%%%%%%%%%% END SCRIPT %%%%%%%%%%%%%%%%%%%% -->
<!-- Create form to contain controls -->
<FORM NAME="Form">
<!-- Create pulldown menu to select which plot to view -->
<P>Select type of plot:
<SELECT NAME=plot_choice>
 <OPTION SELECTED VALUE=first>Line</OPTION>
 <OPTION VALUE=second>Peaks</OPTION>
 <OPTION VALUE=third>Logo</OPTION>
</SELECT>
<!-- Create button to create plot and fill text area -->
<P>Create figure:
<INPUT TYPE="button" NAME="plot_but" VALUE="Plot">


<!-- %%%%%%%%%%%%%%%%%%% BEGIN SCRIPT %%%%%%%%%%%%%%%%%%%% -->
<SCRIPT FOR="plot_but" EVENT="onClick" LANGUAGE="VBScript">
<!-- Start script
Dim plot_choice
Dim text_str    'string to display in text area
Dim form_var 'form object variable
Set form_var = Document.Form
plot_choice = form_var.plot_choice.value
' Condition MATLAB command to execute based on plot choice
If plot_choice = "first" Then
   MLcmd = "figure; plot(1:10);"
   text_str = "Simple line plot of 1 to 10"
   Call MatLab.PutCharArray("text","base",text_str)
Elseif plot_choice = "second" Then
   MLcmd = "figure; mesh(peaks);"
   text_str = "Mesh plot of peaks"
   Call MatLab.PutCharArray("text","base",text_str)
Elseif plot_choice = "third" Then
   MLcmd = "figure; logo;"
   text_str = "MATLAB logo"
   Call MatLab.PutCharArray("text","base",text_str)
End If
```

```
' Execute command in MATLAB
MatLab.execute(MLcmd)
' Get variable from MATLAB into VBScript
Call MatLab.GetWorkspaceData("text","base","text_str")
' Update text area
form_var.plottext.value = text_str
' End script -->
</SCRIPT>

<!-- %%%%%%%%%%%%%%%%%% END SCRIPT %%%%%%%%%%%%%%%%%% -->
<!-- Create text area to show text -->
<P><TEXTAREA NAME="plottext" ROWS="1" COLS="50"
CONTENTEDITABLE="false"></TEXTAREA>
</FONT>
</FORM>
</BODY>
</HTML>
```

Example — Calling MATLAB Software from a C# Client

This example creates data in the client C# program and passes it to MATLAB. The matrix (containing complex data) is then passed back to the C# program.

The reference to the MATLAB Type Library for C# is:

```
MLApp.MLAppClass matlab = new MLApp.MLAppClass();
```

Here is the complete example:

```
using System;
namespace ConsoleApplication4
{
class Class1
{
[STAThread]
static void Main(string[] args)
{
MLApp.MLAppClass matlab = new MLApp.MLAppClass();

System.Array pr = new double[4];
```

```
pr.SetValue(11,0);
pr.SetValue(12,1);
pr.SetValue(13,2);
pr.SetValue(14,3);

System.Array pi = new double[4];
pi.SetValue(1,0);
pi.SetValue(2,1);
pi.SetValue(3,2);
pi.SetValue(4,3);

matlab.PutFullMatrix("a", "base", pr, pi);

System.Array prresult = new double[4];
System.Array piresult = new double[4];

matlab.GetFullMatrix("a", "base", ref prresult, ref piresult);
}
}
}
```

11

Web Services in MATLAB Applications

- "Product Overview" on page 11-2
- "Using Web Services in MATLAB Applications" on page 11-7
- "Building MATLAB Applications with Web Services" on page 11-11

Product Overview

Introduction

The term *Web service* encompasses a set of XML-based technologies for making remote procedure calls over a network. The network can be a local intranet within an organization or a remote server on the other side of the globe. In short, Web services let applications running on disparate operating systems and development platforms communicate with each other.

MATLAB software acts as a Web service client by sending requests to a server and handling the responses. MATLAB implements the following Web service technologies:

- Simple Object Access Protocol (SOAP)
- Web Services Description Language (WSDL)

SOAP defines a standard for making XML-based exchanges between clients and servers. The client initiates the client/server interaction, which usually takes place over HTTP. When the server receives the request, which includes the operation to be performed and any necessary parameters, it sends back a response.

Web Service Examples

The following example shows a simple HTTP-based SOAP request for retrieving the local temperature by zip code:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soapenv:Body>
     <ns1:getTemp
         xmlns:ns1="urn:xmethods-Temperature-Demo"
         soapenv:encodingStyle=
           "http://schemas.xmlsoap.org/soap/encoding/">
       <zipcode xsi:type="xsd:string">94041</zipcode>
     </ns1:getTemp>
   </soapenv:Body>
 </soapenv:Envelope>
```

The SOAP protocol defines an envelope, and inside the envelope, defines
a message body. Also, inside the message body, the SOAP method
`getTempRequest` is specified, as well as the `zipcode` parameter.

In the response sent by the server, notice that the SOAP message structure is
similar:

```
 <?xml version="1.0" encoding="UTF-8"?>
 <soapenv:Envelope
     xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soapenv:Body>
     <ns1:getTempResponse
         xmlns:ns1="urn:xmethods-Temperature-Demo"
         soapenv:encodingStyle=
           "http://schemas.xmlsoap.org/soap/encoding/">
       <return xsi:type="xsd:float">68.0</return>
     </ns1:getTempResponse>
   </soapenv:Body>
 </soapenv:Envelope>
```

In the code, SOAP defines the envelope and message body as well as the
response (`return`).

Most SOAP implementations use WSDL, an XML-based language, to describe
and locate available services. The following example shows the message and

service definitions of the WSDL file for the temperature service from the previous examples:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="TemperatureService"
    targetNamespace=
      "http://www.xmethods.net/sd/TemperatureService.wsdl"
    xmlns:tns=
      "http://www.xmethods.net/sd/TemperatureService.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="getTempRequest">
   <part name="zipcode" type="xsd:string"/>
  </message>
  <message name="getTempResponse">
    <part name="return" type="xsd:float"/>
  </message>
  <portType name="TemperaturePortType">
    <operation name="getTemp">
      <input message="tns:getTempRequest"/>
      <output message="tns:getTempResponse"/>
    </operation>
  </portType>
  <binding name="TemperatureBinding"
      type="tns:TemperaturePortType">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getTemp">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
            namespace="urn:xmethods-Temperature-Demo"
            encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
            namespace="urn:xmethods-Temperature-Demo"
            encodingStyle=
```

```
                    "http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
  <service name="TemperatureService">
    <documentation>Returns current temperature in a given U.S.
      zipcode</documentation>
    <port name="TemperaturePort"
        binding="tns:TemperatureBinding">
      <soap:address
        location=
        "http://services.xmethods.net:80/soap/servlet/rpcrouter"/>
    </port>
  </service>
</definitions>
```

The code defines the request and response message actions (getTempRequest
and getTempResponse) and the service name (TemperatureService).

Understanding Data Type Conversions

Using SOAP data types, MATLAB automatically converts XML types to
native MATLAB types and vice versa. The following table contains the XML
type with the corresponding MATLAB type.

XML Data Type	MATLAB Data Type
string	char array
boolean	logical scalar
decimal	double scalar
float	double scalar
double	double scalar
duration	double scalar
time	double scalar
date	double scalar
gYearMonth	char array

XML Data Type	MATLAB Data Type
gYear	char array
gMonthDay	char array
hexbinary	double array
base64Binary	double array
anyURI	char array
QName	char array

Finding More Information About Web Services

To learn more about SOAP, see the following resources:

- World Wide Web Consortium (W3C®) SOAP specification

- Apache Axis Web Services

- W3 Schools SOAP Tutorial

To learn more about WSDL, see the following resources:

- W3C WSDL specification

- WSDL4J Project

- W3 Schools WSDL Tutorial

To find publicly available Web services and for more information about popular development platforms for Web services, see the following resources:

- XMethods

- Sun Java Web Services

- Microsoft Developer Network—Web Services

Using Web Services in MATLAB Applications

In this section...
"Getting Started" on page 11-7
"Building a Simple Web Service" on page 11-7

Getting Started

Use the `createClassFromWsdl` function to call Web service methods. The function creates a MATLAB class based on the methods of the Web service application program interface (API).

Here is an example of the `createClassFromWsdl` function using a URL:

```
createClassFromWsdl('http://example.com/service.wsdl')
```

The example passes a URL to a WSDL file to the function. The following example uses a file path instead of a URL:

```
createClassFromWsdl('\myservicedirectory\service.wsdl')
```

The example passes a relative file path to the function. Keep in mind that the target file must contain WSDL.

Note To call remote Web services with MATLAB, you must have a working Internet connection.

Building a Simple Web Service

The following procedure walks you through the necessary steps to build a simple Web service. To begin, the procedure shows you how to find the Currency Exchange Rate Web service:

1 In a Web browser, go to the XMethods Web site at http://www.xmethods.net/.

2 In **XMethods Demo Services**, click the **Currency Exchange Rate** link near the bottom of the page.

3 On the Currency Exchange Rate Web page, find the WSDL URL, as well as links to analyze the WSDL. Click the **View RPC Profile** link.

In the RPC Profile page, find the available methods. In this case, the available method is `getRate`.

In addition to the method name, notice the input and output parameters and their data types. The output parameter returns a `float` data type. In "Understanding Data Type Conversions" on page 11-5, note that MATLAB converts `float` to a `double` scalar.

4 Enter the following code at the MATLAB command line to pass the WSDL URL to the function`createClassFromWsdl`, creating the `CurrencyExchangeService` class:

```
createClassFromWsdl(['http://www.xmethods.net/sd/2001' ...
                     '/CurrencyExchangeService.wsdl']);
ans =
    CurrencyExchangeService
```

Use the MATLAB `methods` function to view the methods associated with the `CurrencyExchangeService` class:

```
methods(CurrencyExchangeService)

   Methods for class CurrencyExchangeService:

   CurrencyExchangeService    getRate    display
```

5 In the current MATLAB directory, find the `@CurrencyExchangeService` folder. In the folder, you see the following files:

- `CurrencyExchangeService.m` — Contains the M-code for MATLAB object constructor
- `display.m` — Contains the M-code for a generic display method
- `getRate.m` — Contains the M-code for the `getRate` method

The `createClassFromWsdl` function automatically creates a file for each Web service method, a file for a generic display method, and a file for the Web service MATLAB object.

You can use the MATLAB `help` function to see the method signature, such as:

```
help CurrencyExchangeService/getRate

getRate(obj,country1,country2)

      Input:
        country1 = (string)
        country2 = (string)

      Output:
        Result = (float)
```

6 Call the `getRate` method to obtain the exchange rate between two countries.

```
ces = CurrencyExchangeService;

getRate(ces, 'USA', 'France')
ans =
    5.1127

getRate(ces, 'Argentina', 'Chile')
ans =
    172.3052
```

To review, the `createClassFromWsdl` function performs the following actions:

- Fetches and parses the WSDL to determine the Web service API

- Creates a folder, such as `@CurrencyExchangeService`, in the current MATLAB directory

- Creates the necessary M files in the directory, such as `getRate.m`, `display.m`, and `CurrencyExchangeService.m`, based on the service API

For more information about object-oriented programming in MATLAB, see the *MATLAB Programming Fundamentals* documentation.

Building MATLAB Applications with Web Services

In this section...
"Understanding Web Service Limitations" on page 11-11
"Programming with Web Services" on page 11-11

Understanding Web Service Limitations

At the time of this writing, Web service technologies continue to evolve and change. The following list contains possible limitations to consider before building MATLAB applications with Web services:

- The majority of Web services are made available via HTTP. Like the Internet itself, quality of service cannot be guaranteed. Therefore, your application performance might suffer or might appear unreliable.

- Web services and the related technologies like WSDL and SOAP are relatively new. As with any new technology, established procedures and best practices are still being written.

- If you plan to call remote Web services, make sure you validate their accuracy and reliability. Also, Web services that are free today might not remain free in the future.

Programming with Web Services

Because the Internet is inherently unpredictable, make sure to take proper precautions in programming with Web services. One way to minimize the risk is to use common program control and error-handling routines.

Common programming techniques you might use include

- `Try - Catch` statements can catch errors that result from method calls as well as creating the MATLAB class from the WSDL. The following example shows a method call in a `try - catch` statement:

```
try
 r = getRate(CurrencyExchangeService, 'USA', 'France');
catch
 r = Nan;
 disp(lasterr);
end
```

- If statements can check that expressions or statements are true or false. For example, if you have a valid URL and WSDL file, such as:

```
wsdlUrl = ['http://www.xmethods.net/sd/2001' ...
           '/CurrencyExchangeService.wsdl'];
wsdlFile = 'CurrencyExchangeService.wsdl';
```

  the following if statement caches the WSDL locally:

```
if ~(exist(wsdlFile,'file') == 2)
    urlwrite(wsdlUrl,wsdlFile);
end
```

- Error functions can be used to throw specific errors. The following example shows an error function used in an try - catch statement:

```
try
 r = getRate(CurrencyExchangeService, 'USA', 'France');
catch
 error('Could not return exchange rate');
end
```

For more information about program control and error-handling statements, see the *MATLAB Programming Fundamentals* documentation.

12

Serial Port I/O

Introduction

In this section...
"What Is the MATLAB Serial Port Interface?" on page 12-2
"Supported Serial Port Interface Standards" on page 12-3
"Supported Platforms" on page 12-3
"Using the Examples with Your Device" on page 12-3

What Is the MATLAB Serial Port Interface?

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

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

Instrument Control Toolbox™ software provides additional serial port functionality. In addition to command-line access, this toolbox has a graphical tool called the Test & Measurement Tool, which allows you to communicate with, configure, and transfer data with your serial device without writing code. The Test & Measurement Tool generates MATLAB code for your serial device that you can later reuse to communicate with your device or to develop GUI-based applications. The toolbox includes additional serial I/O utility functions that facilitate object creation and configuration, instrument communication, and so on. With the toolbox you can communicate with GPIB- or VISA-compatible instruments.

For more information, see the Instrument Control Toolbox documentation.

If you want to communicate with PC-compatible data acquisition hardware such as multifunction I/O boards, you need Data Acquisition Toolbox™ software.

For more information, see the Data Acquisition Toolbox documentation.

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

Supported Serial Port Interface Standards

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

This guide assumes you are using the RS-232 standard, discussed in "Overview of the Serial Port" on page 12-4. Refer to your computer and device documentation to see which interface standard you can use.

Supported Platforms

The MATLAB serial port interface is supported on Microsoft Windows 32-bit, Linux[23] 32-bit, and Sun Solaris 64-bit platforms.

Using the Examples with Your Device

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

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

---

23. Linux is a registered trademark of Linus Torvalds.

Overview of the Serial Port

Introduction

For many serial port applications, you can communicate with your device without detailed knowledge of how the serial port works. If your application is straightforward, or if you are already familiar with the previously mentioned topics, you might want to begin with "The Serial Port Session" on page 12-18 to see how to use your serial port device with MATLAB software.

What Is Serial Communication?

Serial communication is the most common low-level protocol for communicating between two or more devices. Normally, one device is a computer, while the other device can be a modem, a printer, another computer, or a scientific instrument such as an oscilloscope or a function generator.

As the name suggests, the serial port sends and receives bytes of information in a serial fashion — one bit at a time. These bytes are transmitted using either a binary (numerical) format or a text format.

The Serial Port Interface Standard

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

The original serial port interface standard was given by RS-232, which stands for Recommended Standard number 232. The term *RS-232* is still in popular use, and is used in this guide when referring to a serial communication port that follows the TIA/EIA-232 standard. RS-232 defines these serial port characteristics:

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

Primary communication is accomplished using three pins: the Transmit Data pin, the Receive Data pin, and the Ground pin. Other pins are available for data flow control, but are not required.

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

Connecting Two Devices with a Serial Cable

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

Throughout this guide, your computer is considered a DTE, while peripheral devices such as modems and printers are considered DCEs. Many scientific instruments function as DTEs.

Because RS-232 mainly involves connecting a DTE to a DCE, the pin assignments are defined such that straight-through cabling is used, where pin 1 is connected to pin 1, pin 2 is connected to pin 2, and so on. The following diagram shows a DTE to DCE serial connection using the transmit data (TD) pin and the receive data (RD) pin.

For more information about serial port pins, see "Serial Port Signals and Pin Assignments" on page 12-6.
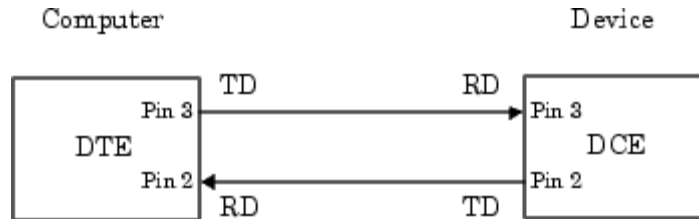
If you connect two DTEs or two DCEs using a straight serial cable, the TD pins on each device are connected to each other, and the RD pins on each device are connected to each other. Therefore, to connect two like devices, you must use a *null modem* cable. As shown in the following diagram, null modem cables cross the transmit and receive lines in the cable.
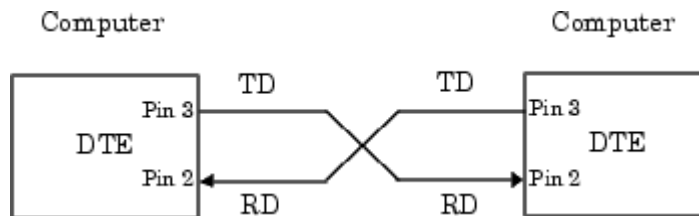
Note You can connect multiple RS-422 or RS-485 devices to a serial port. If you have an RS-232/RS-485 adaptor, you can use the MATLAB serial port object with these devices.

Serial Port Signals and Pin Assignments

Serial ports consist of two signal types: data signals and control signals. To support these signal types, as well as the signal ground, the RS-232 standard defines a 25-pin connection. However, most Windows and UNIX[24] platforms use a 9-pin connection. In fact, only three pins are required for serial port

24. UNIX is a registered trademark of The Open Group in the United States and other countries.

communications: one for receiving data, one for transmitting data, and one for the signal ground.

The following diagram shows the pin assignment scheme for a 9-pin male connector on a DTE.
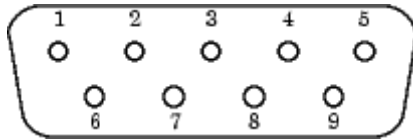
The pins and signals associated with the 9-pin connector are described in the following table. Refer to the RS-232 standard for a description of the signals and pin assignments used for a 25-pin connector.

Serial Port Pin and Signal Assignments

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

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

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

Signal States

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

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

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

The following diagram shows the on and off states for a data signal and for a control signal.
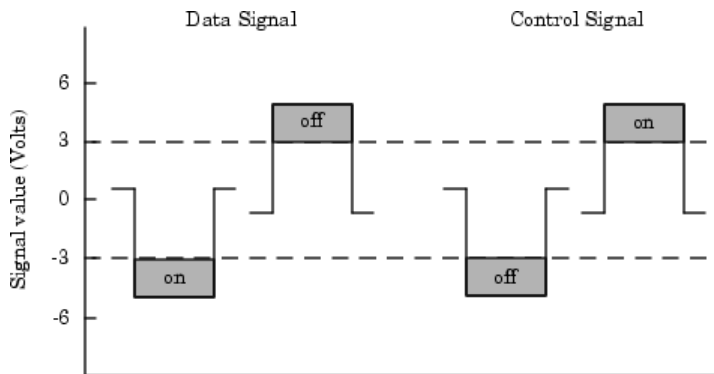
The Data Pins

Most serial port devices support *full-duplex* communication meaning that they can send and receive data at the same time. Therefore, separate pins are used for transmitting and receiving data. For these devices, the TD, RD, and GND pins are used. However, some types of serial port devices support only one-way or *half-duplex* communications. For these devices, only the TD and GND pins are used. This guide assumes that a full-duplex serial port is connected to your device.

The TD pin carries data transmitted by a DTE to a DCE. The RD pin carries data that is received by a DTE from a DCE.

The Control Pins

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

• "The RTS and CTS Pins" on page 12-9

• "The DTR and DSR Pins" on page 12-10

• "The CD and RI Pins" on page 12-10

The RTS and CTS Pins. The RTS and CTS pins are used to signal whether the devices are ready to send or receive data. This type of data flow control—called *hardware handshaking*—is used to prevent data loss during transmission. When enabled for both the DTE and DCE, hardware handshaking using RTS and CTS follows these steps:

1 The DTE asserts the RTS pin to instruct the DCE that it is ready to receive data.

2 The DCE asserts the CTS pin indicating that it is clear to send data over the TD pin. If data can no longer be sent, the CTS pin is unasserted.

3 The data is transmitted to the DTE over the TD pin. If data can no longer be accepted, the RTS pin is unasserted by the DTE and the data transmission is stopped.

To enable hardware handshaking in MATLAB software, see "Controlling the Flow of Data: Handshaking" on page 12-62.

The DTR and DSR Pins. Many devices use the DSR and DTR pins to signal if they are connected and powered. Signaling the presence of connected devices using DTR and DSR follows these steps:

1 The DTE asserts the DTR pin to request that the DCE connect to the communication line.

2 The DCE asserts the DSR pin to indicate it is connected.

3 DCE unasserts the DSR pin when it is disconnected from the communication line.

The DTR and DSR pins were originally designed to provide an alternative method of hardware handshaking. However, the RTS and CTS pins are usually used in this way, and not the DSR and DTR pins. Refer to your device documentation to determine its specific pin behavior.

The CD and RI Pins. The CD and RI pins are typically used to indicate the presence of certain signals during modem-modem connections.

A modem uses a CD pin to signal that it has made a connection with another modem, or has detected a carrier tone. CD is asserted when the DCE is receiving a signal of a suitable frequency. CD is unasserted if the DCE is not receiving a suitable signal.

The RI pin is used to indicate the presence of an audible ringing signal. RI is asserted when the DCE is receiving a ringing signal. RI is unasserted when the DCE is not receiving a ringing signal (e.g., it is between rings).

Serial Data Format

The serial data format includes one start bit, between five and eight data bits, and one stop bit. A parity bit and an additional stop bit might be included in the format as well. The following diagram illustrates the serial data format.

Start bit   Data bits   Parity bit   Stop bits
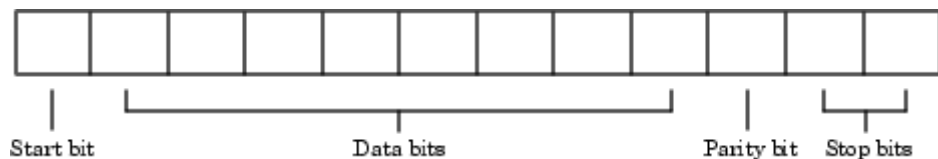
The following notation expresses the format for serial port data:

number of data bits - parity type - number of stop bits

For example, 8-N-1 is interpreted as eight data bits, no parity bit, and one stop bit, while 7-E-2 is interpreted as seven data bits, even parity, and two stop bits.

The data bits are often referred to as a *character* because these bits usually represent an ASCII character. The remaining bits are called *framing bits* because they frame the data bits.

Bytes Versus Values

A *byte* is the collection of bits that comprise the serial data format. At first, this term might seem inaccurate because a byte is 8 bits and the serial data format can range between 7 bits and 12 bits. However, when serial data is stored on your computer, the framing bits are stripped away, and only the data bits are retained. Moreover, eight data bits are always used regardless of the number of data bits specified for transmission, with the unused bits assigned a value of 0.

When reading or writing data, you might need to specify a *value*, which can consist of one or more bytes. For example, if you read one value from a device using the int32 format, that value consists of four bytes. For more information about reading and writing values, see "Writing and Reading Data" on page 12-31.

Synchronous and Asynchronous Communication

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

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

faster data transfer rates than asynchronous methods, because additional bits to mark the beginning and end of each data byte are not required.

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

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

Note When used in this guide, the terms *synchronous* and *asynchronous* refer to whether read or write operations block access to the MATLAB command line. For more information, see "Controlling Access to the MATLAB Command Line" on page 12-32.

How Are the Bits Transmitted?

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

1 The start bit is transmitted with a value of 0.

2 The data bits are transmitted. The first data bit corresponds to the least significant bit (LSB), while the last data bit corresponds to the most significant bit (MSB).

3 The parity bit (if defined) is transmitted.

4 One or two stop bits are transmitted, each with a value of 1.

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

Start and Stop Bits

As described in "Synchronous and Asynchronous Communication" on page 12-11, most serial ports operate asynchronously. This means that the transmitted byte must be identified by start and stop bits. The start bit indicates when the data byte is about to begin; the stop bit(s) indicate(s) when the data byte has been transferred. The process of identifying bytes with the serial data format follows these steps:

1 When a serial port pin is idle (not transmitting data), it is in an on state.

2 When data is about to be transmitted, the serial port pin switches to an off state due to the start bit.

3 The serial port pin switches back to an on state due to the stop bit(s). This indicates the end of the byte.

Data Bits

The data bits transferred through a serial port might represent device commands, sensor readings, error messages, and so on. The data can be transferred as either binary data or ASCII data.

Most serial ports use between five and eight data bits. Binary data is typically transmitted as eight bits. Text-based data is transmitted as either seven bits or eight bits. If the data is based on the ASCII character set, a minimum of seven bits is required because there are 2^7 or 128 distinct characters. If an eighth bit is used, it must have a value of 0. If the data is based on the extended ASCII character set, eight bits must be used because there are 2^8 or 256 distinct characters.

The Parity Bit

The parity bit provides simple error (parity) checking for the transmitted data. The following table shows the types of parity checking.

Parity Types

Parity Type	Description
Even	The data bits plus the parity bit result in an even number of 1s.

Parity Types (Continued)

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

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

The parity checking process follows these steps:

1 The transmitting device sets the parity bit to 0 or to 1, depending on the data bit values and the type of parity-checking selected.

2 The receiving device checks if the parity bit is consistent with the transmitted data. If it is, the data bits are accepted. If it is not, an error is returned.

Note Parity checking can detect only 1-bit errors. Multiple-bit errors can appear as valid data.

For example, suppose the data bits 01110001 are transmitted to your computer. If even parity is selected, the parity bit is set to 0 by the transmitting device to produce an even number of 1s. If odd parity is selected, the parity bit is set to 1 by the transmitting device to produce an odd number of 1s.

Finding Serial Port Information for Your Platform

This section describes the ways to find serial port information for Windows and UNIX platforms.

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

Microsoft Windows Platform

You can access serial port information through the **System Properties** dialog. To access this on a Windows XP platform,

1 Right-click **My Computer** on the desktop, and select **Properties**.

2 In the **System Properties** dialog, click the **Hardware** tab.

3 Click **Device Manager**.

4 In the **Device Manager** dialog, expand the Ports node.

5 Double-click the `Communications Port (COM1)` node.

6 Select the **Port Settings** tab.

MATLAB displays the following Ports dialog box.
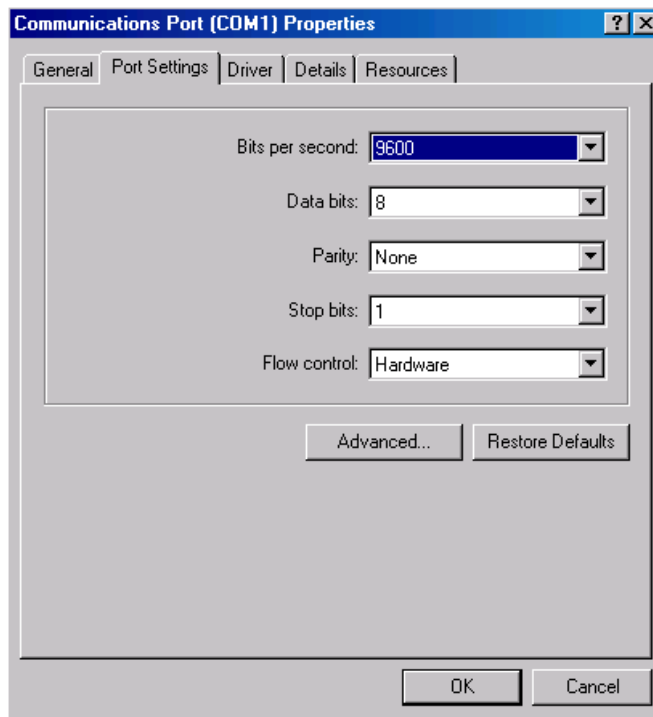
UNIX Platform

To find serial port information for UNIX platforms, you need to know the serial port names. These names might vary between different operating systems.

On a Linux platform, serial port devices are typically named ttyS0, ttyS1, etc. Use the setserial command to display or configure serial port information. For example, to display which ports are available:

```
setserial -bg /dev/ttyS*
/dev/ttyS0 at 0x03f8 (irq = 4) is a 16550A
/dev/ttyS1 at 0x02f8 (irq = 3) is a 16550A
```

To display detailed information about `ttyS0`:

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

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

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

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

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

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

Selected Bibliography

[1] TIA/EIA-232-F, *Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*.

[2] Jan Axelson, *Serial Port Complete*, Lakeview Research, Madison, WI, 1998.

[3] *Instrument Communication Handbook*, IOTech, Inc., Cleveland, OH, 1991.

[4] *TDS 200-Series Two Channel Digital Oscilloscope Programmer Manual,* Tektronix, Inc., Wilsonville, OR.

[5] *Courier High Speed Modems User's Manual*, U.S. Robotics, Inc., Skokie, IL, 1994.

Getting Started with Serial I/O

In this section...
"Example: Getting Started" on page 12-18
"The Serial Port Session" on page 12-18
"Configuring and Returning Properties" on page 12-20

Example: Getting Started

This example illustrates some basic serial port commands.

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

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

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

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

The Serial Port Session

This example describes the steps you use to perform any serial port task from beginning to end.

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

1 Create a serial port object — Create a serial port object for a specific serial port using the `serial` creation function.

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

2 Connect to the device — Connect the serial port object to the device using the `fopen` function.

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

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

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

4 Write and read data — Write data to the device using the `fprintf` or `fwrite` function, and read data from the device using the `fgetl`, `fgets`, `fread`, `fscanf`, or `readasync` function.

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

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

The serial port session is reinforced in many of the serial port documentation examples. To see a basic example that uses the steps shown above, see "Example: Getting Started" on page 12-18.

Configuring and Returning Properties

This example describes how you display serial port property names and property values, and how you assign values to properties.

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

Displaying Property Names and Property Values

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

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

    SERIAL specific properties:
    BaudRate
    BreakInterruptFcn
    DataBits
    DataTerminalReady: [ {on} | off ]
    FlowControl: [ {none} | hardware | software ]
```

```
Parity: [ {none} | odd | even | mark | space ]
PinStatusFcn
Port
ReadAsyncMode: [ {continuous} | manual ]
RequestToSend: [ {on} | off ]
StopBits
Terminator
```

Use the get function to display one or more properties and their current values to the command line. To display all properties and their current values:

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

    SERIAL specific properties:
    BaudRate = 9600
```

```
BreakInterruptFcn =
DataBits = 8
DataTerminalReady = on
FlowControl = none
Parity = none
PinStatus = [1x1 struct]
PinStatusFcn =
Port = COM1
ReadAsyncMode = continuous
RequestToSend = on
StopBits = 1
Terminator = LF
```

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

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

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

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

Use the dot notation to display a single property value.

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

Configuring Property Values

You can configure property values using the set function:

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

or the dot notation:

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

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

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

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

In practice, you can configure many of the properties at any time while the serial port object exists — including during object creation. However, some properties are not configurable while the object is connected to the device or when recording information to disk. For information about when a property is configurable, see "Property Reference" on page 12-75.

Specifying Property Names

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

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

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

Default Property Values

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

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

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

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

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

Creating a Serial Port Object

In this section...
"Overview of a Serial Port Object" on page 12-25
"Configuring Properties During Object Creation" on page 12-26
"The Serial Port Object Display" on page 12-26
"Creating an Array of Serial Port Objects" on page 12-27

Overview of a Serial Port Object

The serial function requires the name of the serial port connected to your device as an input argument. Additionally, you can configure property values during object creation. For example, to create a serial port object associated with the serial port COM1, enter:

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

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

```
whos s
  Name      Size           Bytes  Class

   s        1x1              512  serial object

Grand total is 11 elements using 512 bytes
```

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

Once the serial port object is created, the following properties are automatically assigned values. These general-purpose properties provide

descriptive information about the serial port object based on the object type and the serial port.

Descriptive General Purpose Properties

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

Display the values of these properties for s with the get function.

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

Configuring Properties During Object Creation

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

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

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

The Serial Port Object Display

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

- Type the serial port object variable name at the command line.

- Exclude the semicolon when creating a serial port object.

- Exclude the semicolon when configuring properties using the dot notation.

The display summary for the serial port object s is:

```
Serial Port Object : Serial-COM1

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

Communication State
   Status:            closed
   RecordStatus:      off

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

Creating an Array of Serial Port Objects

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

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

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

```
x = [s1 s2]

Instrument Object Array

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

To create the column array y, enter:

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

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

```
z = [s1 s2;s1 s2];
??? Error using ==> serial/vertcat
Only a row or column vector of instrument objects can be created.
```

Depending on your application, you might want to pass an array of serial port objects to a function. For example, to configure the baud rate and parity for s1 and s2 using one call to set:

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

To see which functions accept a serial port object array as an input, see the Serial Port Devices functional reference.

Connecting to the Device

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

```
fopen(s)
```

Some properties are read only while the serial port object is connected and must be configured before using `fopen`. Examples include the `InputBufferSize` and the `OutputBufferSize` properties. To determine when you can configure a property, see "Property Reference" on page 12-75.

Note You can create any number of serial port objects, but you can connect only one serial port object per MATLAB session to a given serial port at a time. However, the serial port is not locked by the session, so other applications or other instances of MATLAB software can access the same serial port, which could result in a conflict, with unpredictable results.

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

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

As shown in the following illustration, the connection between the serial port object and the device is complete; data is readable and writable.
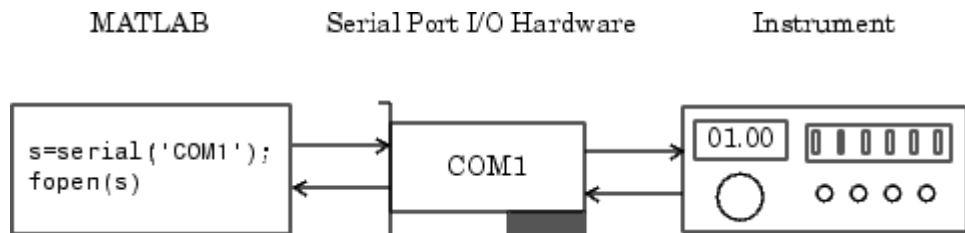
Configuring Communication Settings

Before you can write or read data, both the serial port object and the device must have identical communication settings. Configuring serial port communications involves specifying values for properties that control the baud rate and the serial data format. The following table describes these properties.

Communication Properties

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

Note If the serial port object and the device communication settings are not identical, data is not readable or writable.

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

Writing and Reading Data

In this section...
"Before You Begin" on page 12-31
"Example — Introduction to Writing and Reading Data" on page 12-31
"Controlling Access to the MATLAB Command Line" on page 12-32
"Writing Data" on page 12-33
"Reading Data" on page 12-38
"Example — Writing and Reading Text Data" on page 12-44
"Example — Parsing Input Data Using strread" on page 12-46
"Example — Reading Binary Data" on page 12-47

Before You Begin

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

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

For write operations, these questions are answered in "Writing Data" on page 12-33. For read operations, these questions are answered in "Reading Data" on page 12-38.

Example — Introduction to Writing and Reading Data

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

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

```
out = fscanf(s)
```

The resulting identification information is:

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

End the serial port session.

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

Controlling Access to the MATLAB Command Line

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

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

The terms *synchronous* and *asynchronous* are often used to describe how the serial port operates at the hardware level. The RS-232 standard supports an asynchronous communication protocol. Using this protocol, each device uses its own internal clock. The data transmission is synchronized using the start bit of the bytes, while one or more stop bits indicate the end of the byte. For more information on start bits and stop bits, see "Serial Data Format" on page 12-10. The RS-232 standard also supports a synchronous mode where all transmitted bits are synchronized to a common clock signal.

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

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

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

- You can issue another command while the write or read function is executing.

- You can use all supported callback properties (see "Events and Callbacks" on page 12-50).

For example, because serial ports have separate read and write pins, you can simultaneously read and write data. This is illustrated in the following diagram.
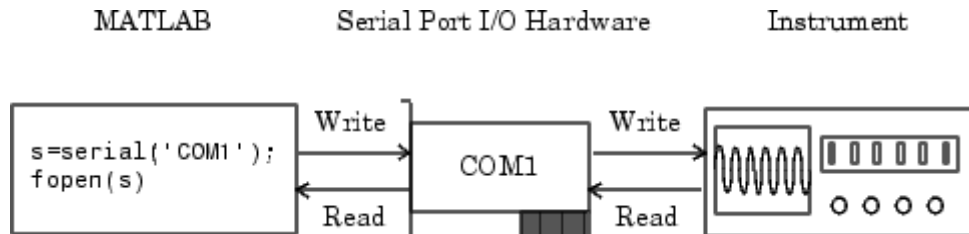
Writing Data

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

- "The Output Buffer and Data Flow" on page 12-34 describes the flow of data from MATLAB software to the device.

- "Writing Text Data" on page 12-36 describes how to write text data (string commands) to the device.

- "Writing Binary Data" on page 12-38 describes how to write binary (numerical) data to the device.

The following table shows the functions associated with writing data.

Functions Associated with Writing Data

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

The following table shows the properties associated with writing data.

Properties Associated with Writing Data

Property Name	Description
BytesToOutput	Number of bytes currently in the output buffer
OutputBufferSize	Size of the output buffer in bytes
Timeout	Waiting time to complete a read or write operation
TransferStatus	Indicate if an asynchronous read or write operation is in progress
ValuesSent	Total number of values written to the device

The Output Buffer and Data Flow

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

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

2 The data in the output buffer is sent to the device.

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

```
s = serial('COM1');
get(s,{'OutputBufferSize','BytesToOutput'})
```

```
ans =
    [512]    [0]
```

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

For example, suppose you write the string command *IDN? to the TDS 210 oscilloscope using the fprintf function. As shown in the following diagram, the string is first written to the output buffer as six values.
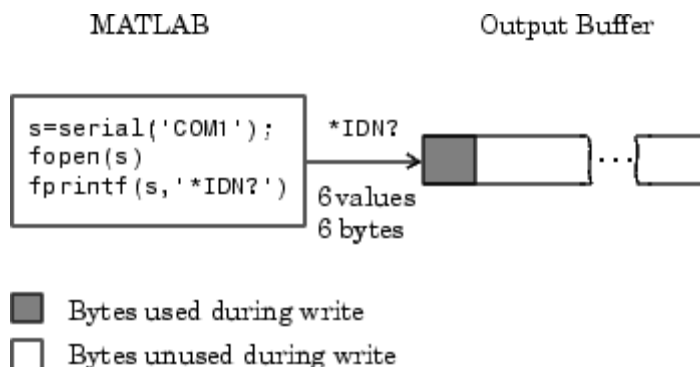
The *IDN? command consists of six values because the terminator is automatically written. Moreover, the default data format for the fprintf function specifies that one value corresponds to one byte. For more information about bytes and values, see "Bytes Versus Values" on page 12-11. fprintf and the terminator are discussed in "Writing Text Data" on page 12-36.

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

Bytes used during write

Bytes unused during write
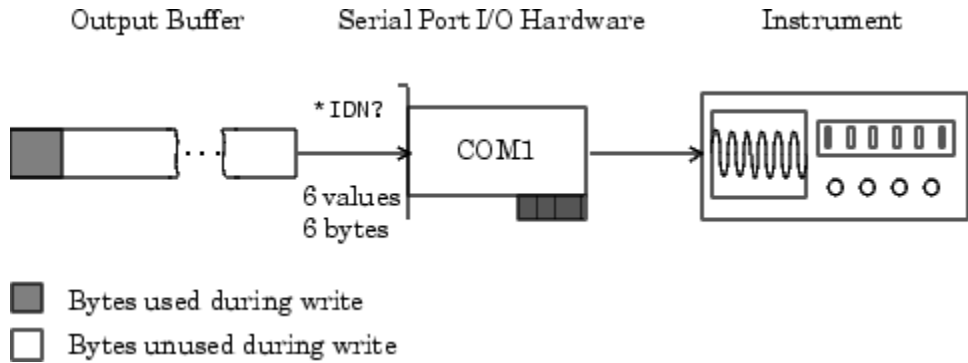
Writing Text Data

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

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

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

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

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

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

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

```
s.Terminator
```

```
ans =
LF
```

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

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

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

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

- Execute an asynchronous read operation because serial ports have separate pins for reading and writing

- Make use of all supported callback properties

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

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

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

- The specified data is written.

- The time specified by the `Timeout` property passes.

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

Writing Binary Data

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

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

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

By default, `fwrite` operates synchronously. To write binary data asynchronously to the device, you must specify `async` as the last input argument to `fwrite`. For more information about synchronous and asynchronous write operations, see "Writing Text Data" on page 12-36. For a description of the rules used by `fwrite` to complete a write operation, refer to its reference pages.

Reading Data

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

- "The Input Buffer and Data Flow" on page 12-39 describes the flow of data from the device to MATLAB software.

- "Reading Text Data" on page 12-41 describes how to read from the device, and format the data as text.

- "Reading Binary Data" on page 12-43 describes how to read binary (numerical) data from the device.

The following table shows the functions associated with reading data.

Functions Associated with Reading Data

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

The following table shows the properties associated with reading data.

Properties Associated with Reading Data

Property Name	Description
BytesAvailable	Number of bytes available in the input buffer
InputBufferSize	Size of the input buffer in bytes
ReadAsyncMode	Specify whether an asynchronous read operation is continuous or manual
Timeout	Waiting time to complete a read or write operation
TransferStatus	Indicate if an asynchronous read or write operation is in progress
ValuesReceived	Total number of values read from the device

The Input Buffer and Data Flow

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

1 The data read from the device is stored in the input buffer.

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

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

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

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

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

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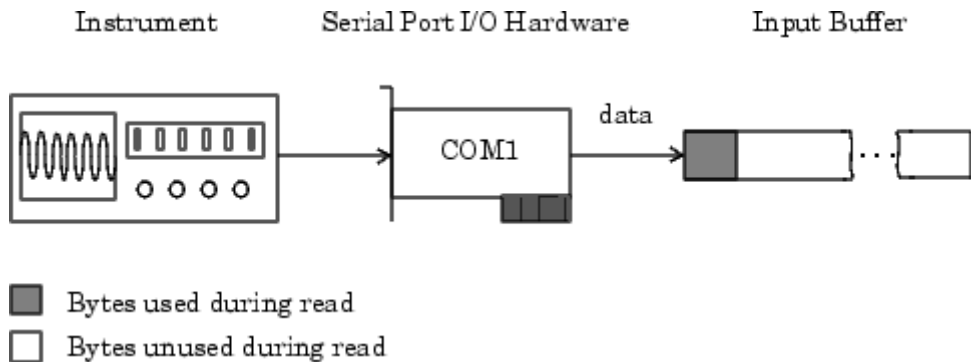
Note that for a given read operation, you might not know the number of bytes returned by the device. Therefore, you might need to preset the `InputBufferSize` property to a sufficiently large value before connecting the serial port object.

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

Input Buffer   MATLAB

data

out=fscanf(s)

■ Bytes used during read

□ Bytes unused during read
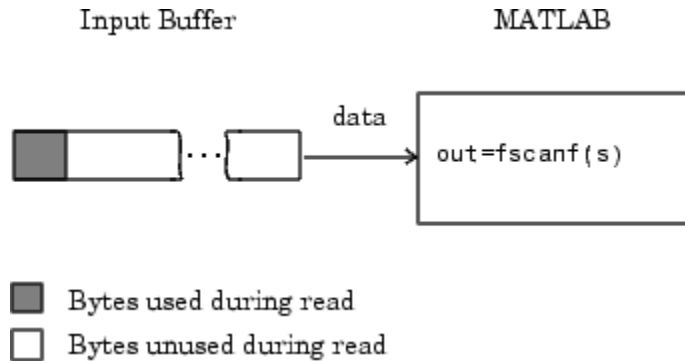
Reading Text Data

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

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

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

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

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

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

Synchronous Versus Asynchronous Read Operations. You specify whether read operations are synchronous or asynchronous with the `ReadAsyncMode` property. You can configure `ReadAsyncMode` to `continuous` or `manual`.

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

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

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

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

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

- Execute an asynchronous write operation because serial ports have separate pins for reading and writing

• Make use of all supported callback properties

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

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

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

• The terminator specified by the `Terminator` property is read.

• The time specified by the `Timeout` property passes.

• The specified number of values specified is read.

• The input buffer is filled.

Reading Binary Data

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

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

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

Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the device. You can verify the number of values read with the `BytesAvailable` property.

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

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

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

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

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

For more information about synchronous and asynchronous read operations, see "Reading Text Data" on page 12-41. For a description of the rules used by `fread` to complete a read operation, refer to its reference pages.

Example — Writing and Reading Text Data

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

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

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

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

2 Connect to the device — Connect `s` to the oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

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

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

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

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

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

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

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

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

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

```
ptop = fscanf(s,'%g')
ptop =
2.0199999809E0
```

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

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

Example — Parsing Input Data Using strread

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

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

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

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

2 Connect to the device — Connect s to the oscilloscope. Because the default value for the ReadAsyncMode property is continuous, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

3 Write and read data — Write the RS232? command to the instrument using fprintf, and then read back the result of the command using fscanf. RS232? queries the RS-232 settings and returns the baud rate, the software flow control setting, the hardware flow control setting, the parity type, and the terminator.

```
fprintf(s,'RS232?')
data = fscanf(s)
data =
9600;0;0;NONE;LF
```

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

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

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

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

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

Example — Reading Binary Data

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

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

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

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

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

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

3 Connect to the device — Connect s to the oscilloscope. Because the default value for the ReadAsyncMode property is continuous, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

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

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

Wait until all the data is sent to the input buffer, and then transfer the data to the MATLAB workspace as unsigned 8-bit integers.

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

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

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

Viewing the Bitmap Data

To view the bitmap data, follow these steps:

1 Open a disk file.

2 Write the data to the disk file.

3 Close the disk file.

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

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

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

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

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

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

The following diagram shows the resulting bitmap image.
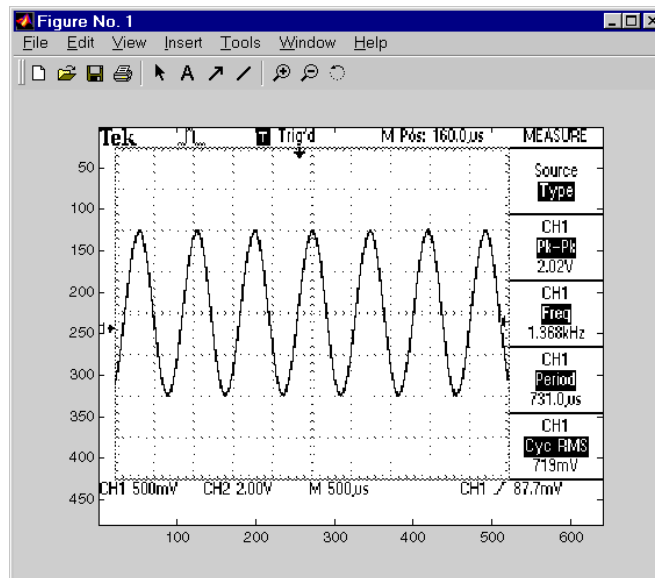
Events and Callbacks

In this section...
"Introduction" on page 12-50
"Example — Introduction to Events and Callbacks" on page 12-50
"Event Types and Callback Properties" on page 12-51
"Responding To Event Information" on page 12-53
"Creating and Executing Callback Functions" on page 12-56
"Enabling Callback Functions After They Error" on page 12-57
"Example — Using Events and Callbacks" on page 12-57

Introduction

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

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

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

Example — Introduction to Events and Callbacks

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

```
s = serial('COM1');
fopen(s)
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @instrcallback;
```

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

MATLAB displays:

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

End the serial port session.

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

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

Event Types and Callback Properties

The following table describes serial port event types and callback properties. This table has two columns and nine rows. In the first column (event type), the second item (bytes available) applies to rows 2 through 4. Also, in the first column the last item (timer) applies to rows 8 and 9.

Event Types and Callback Properties

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

Break-Interrupt Event

A break-interrupt event is generated immediately after a break interrupt is generated by the serial port. The serial port generates a break interrupt when the received data has been in an inactive state longer than the transmission time for one character.

This event executes the callback function specified for the `BreakInterruptFcn` property. It can be generated for both synchronous and asynchronous read and write operations.

Bytes-Available Event

A bytes-available event is generated immediately after a predetermined number of bytes are available in the input buffer or a terminator is read, as determined by the `BytesAvailableFcnMode` property.

If `BytesAvailableFcnMode` is `byte`, the bytes-available event executes the callback function specified for the `BytesAvailableFcn` property every time the number of bytes specified by `BytesAvailableFcnCount` is stored in the input buffer. If `BytesAvailableFcnMode` is `terminator`, the callback function executes every time the character specified by the `Terminator` property is read.

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

Error Event

An error event is generated immediately after an error occurs.

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

An error event is generated when a time-out occurs. A time-out occurs if a read or write operation does not successfully complete within the time specified by the `Timeout` property. An error event is not generated for configuration errors such as setting an invalid property value.

Output-Empty Event

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

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

Pin Status Event

A pin status event is generated immediately after the state (pin value) changes for the CD, CTS, DSR, or RI pins. For a description of these pins, see "Serial Port Signals and Pin Assignments" on page 12-6.

This event executes the callback function specified for the `PinStatusFcn` property. It can be generated for both synchronous and asynchronous read and write operations.

Timer Event

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

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

Responding To Event Information

You can respond to event information in a callback function or in a record file. Event information is stored in a callback function using two fields: `Type` and `Data`. The `Type` field contains the event type, while the `Data` field contains event-specific information. As described in "Creating and Executing Callback Functions" on page 12-56, these two fields are associated with a structure that you define in the callback function header. To learn about recording data and event information to a record file, see "Debugging: Recording Information to Disk" on page 12-65.

The following table shows event types and the values for the `Type` and `Data` fields. The table has three columns and 15 rows. Items in the first column (event type) span several rows, as follows:

Break interrupt: rows 1 and 2

Bytes available: rows 3 and 4

Error: rows 5 through 7

Output empty: rows 8 and 9

Pin status: rows 10 through 13

Timer: rows 14 and 15

Event Information

Event Type	Field	Field Value
Break interrupt	`Type`	`BreakInterrupt`
	`Data.AbsTime`	day-month-year hour:minute:second
Bytes available	`Type`	`BytesAvailable`
	`Data.AbsTime`	day-month-year hour:minute:second
Error	`Type`	`Error`
	`Data.AbsTime`	day-month-year hour:minute:second
	`Data.Message`	An error string
Output empty	`Type`	`OutputEmpty`
	`Data.AbsTime`	day-month-year hour:minute:second

Event Information (Continued)

Event Type	Field	Field Value
Pin status	Type	PinStatus
	Data.AbsTime	day-month-year hour:minute:second
	Data.Pin	CarrierDetect, ClearToSend, DataSetReady, or RingIndicator
	Data.PinValue	on or off
Timer	Type	Timer
	Data.AbsTime	day-month-year hour:minute:second

The following topics describe the Data field values.

The AbsTime Field

The AbsTime field, defined for all events, is the absolute time the event occurred. The absolute time is returned using the clock format: day-month-year hour:minute:second.

The Pin Field

The pin status event uses the Pin field to indicate if the CD, CTS, DSR, or RI pins changed state. For a description of these pins, see "Serial Port Signals and Pin Assignments" on page 12-6.

The PinValue Field

The pin status event uses the PinValue field to indicate the state of the CD, CTS, DSR, or RI pins. Possible values are on or off.

The Message Field

The error event uses the Message field to store the descriptive message that is generated when an error occurs.

Creating and Executing Callback Functions

You can specify the callback function to be executed when a specific event type occurs by including the name of the M-file as the value for the associated callback property. You can specify the callback function as a function handle or as a string cell array element. Function handles are described in the function_handle reference pages.

For example, to execute the callback function mycallback every time the terminator is read from your device:

```
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @mycallback;
```

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

```
s.BytesAvailableFcn = {'mycallback'};
```

M-file callback functions require at least two input arguments. The first argument is the serial port object. The second argument is a variable that captures the event information shown in the table, Event Information on page 12-54. This event information pertains only to the event that caused the callback function to execute. The function header for mycallback is:

```
function mycallback(obj,event)
```

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

```
time = datestr(now,0);
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = {@mycallback,time};
```

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

```
s.BytesAvailableFcn = {'mycallback',time};
```

The corresponding function header is:

```
function mycallback(obj,event,time)
```

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

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

Enabling Callback Functions After They Error

If an error occurs while a callback function is executing the following occurs:

- The callback function is automatically disabled.

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

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

Example — Using Events and Callbacks

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

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

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

2 Connect to the device — Connect `s` to the Tektronix TDS 210 oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

3 Configure properties — Configure s to execute the callback function instrcallback when a bytes-available event or an output-empty event occurs. Because instrcallback requires the serial port object and event information to be passed as input arguments, the callback function is specified as a function handle.

```
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @instrcallback;
s.OutputEmptyFcn = @instrcallback;
```

4 Write and read data — Write the RS232? command asynchronously to the oscilloscope. This command queries the RS-232 settings and returns the baud rate, the software flow control setting, the hardware flow control setting, the parity type, and the terminator.

```
fprintf(s,'RS232?','async')
```

instrcallback is called after the RS232? command is sent, and when the terminator is read. The resulting displays are:

```
OutputEmpty event occurred at 17:37:21 for the object:
Serial-COM1.

BytesAvailable event occurred at 17:37:21 for the object:
Serial-COM1.
```

Read the data from the input buffer.

```
out = fscanf(s)
out =
9600;0;0;NONE;LF
```

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

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

Using Control Pins

In this section...
"Properties of Serial Port Control Pins" on page 12-59
"Signaling the Presence of Connected Devices" on page 12-59
"Controlling the Flow of Data: Handshaking" on page 12-62

Properties of Serial Port Control Pins

As described in "Serial Port Signals and Pin Assignments" on page 12-6, 9-pin serial ports include six control pins. The following table shows properties associated with the serial port control pins.

Control Pin Properties

Property Name	Description
DataTerminalReady	State of the DTR pin
FlowControl	Data flow control method to use
PinStatus	State of the CD, CTS, DSR, and RI pins
RequestToSend	State of the RTS pin

Signaling the Presence of Connected Devices

DTEs and DCEs often use the CD, DSR, RI, and DTR pins to indicate whether a connection is established between serial port devices. Once the connection is established, you can begin to write or read data.

You can monitor the state of the CD, DSR, and RI pins with the PinStatus property. You can specify or monitor the state of the DTR pin with the DataTerminalReady property.

The following example illustrates how these pins are used when two modems are connected to each other.

Example — Connecting Two Modems

This example connects two modems to each other via the same computer, and illustrates how you can monitor the communication status for the computer-modem connections, and for the modem-modem connection. The first modem is connected to COM1, while the second modem is connected to COM2.

1 Create the serial port objects — After the modems are powered on, the serial port object s1 is created for the first modem, and the serial port object s2 is created for the second modem.

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

2 Connect to the devices — s1 and s2 are connected to the modems. Because the default value for the ReadAsyncMode property is continuous, data is asynchronously returned to the input buffers as soon as it is available from the modems.

```
fopen(s1)
fopen(s2)
```

Because the default DataTerminalReady property value is on, the computer (data terminal) is now ready to exchange data with the modems. You can verify that the modems (data sets) can communicate with the computer by examining the value of the Data Set Ready pin with the PinStatus property.

```
s1.Pinstatus
ans =
    CarrierDetect: 'off'
      ClearToSend: 'on'
     DataSetReady: 'on'
    RingIndicator: 'off'
```

The value of the DataSetReady field is on because both modems were powered on before they were connected to the objects.

3 Configure properties — Both modems are configured for a baud rate of 2400 bits per second and a carriage return (CR) terminator.

```
s1.BaudRate = 2400;
s1.Terminator = 'CR';
s2.BaudRate = 2400;
s2.Terminator = 'CR';
```

4 Write and read data — Write the `atd` command to the first modem. This command puts the modem "off the hook," which is equivalent to manually lifting a phone receiver.

```
fprintf(s1,'atd')
```

Write the `ata` command to the second modem. This command puts the modem in "answer mode," which forces it to connect to the first modem.

```
fprintf(s2,'ata')
```

After the two modems negotiate their connection, verify the connection status by examining the value of the Carrier Detect pin using the `PinStatus` property.

```
s1.PinStatus
ans =
    CarrierDetect: 'on'
      ClearToSend: 'on'
     DataSetReady: 'on'
    RingIndicator: 'off'
```

Verify the modem-modem connection by reading the descriptive message returned by the second modem.

```
s2.BytesAvailable
ans =
    25
out = fread(s2,25);
char(out)'
ans =
ata
CONNECT 2400/NONE
```

Now break the connection between the two modems by configuring the `DataTerminalReady` property to `off`. You can verify that the modems are disconnected by examining the Carrier Detect pin value.

```
s1.DataTerminalReady = 'off';
s1.PinStatus
ans =
    CarrierDetect: 'off'
      ClearToSend: 'on'
     DataSetReady: 'on'
    RingIndicator: 'off'
```

5 Disconnect and clean up — Disconnect the objects from the modems and remove the objects from memory and from the MATLAB workspace.

```
fclose([s1 s2])
delete([s1 s2])
clear s1 s2
```

Controlling the Flow of Data: Handshaking

Data flow control or *handshaking* is a method used for communicating between a DCE and a DTE to prevent data loss during transmission. For example, suppose your computer can receive only a limited amount of data before it must be processed. As this limit is reached, a handshaking signal is transmitted to the DCE to stop sending data. When the computer can accept more data, another handshaking signal is transmitted to the DCE to resume sending data.

If supported by your device, you can control data flow using one of these methods:

- Hardware handshaking
- Software handshaking

Note Although you might be able to configure your device for both hardware handshaking and software handshaking at the same time, MATLAB does not support this behavior.

You can specify the data flow control method with the FlowControl property. If FlowControl is hardware, hardware handshaking is used to control data

flow. If `FlowControl` is `software`, software handshaking is used to control data flow. If `FlowControl` is `none`, no handshaking is used.

Hardware Handshaking

Hardware handshaking uses specific serial port pins to control data flow. In most cases, these are the RTS and CTS pins. Hardware handshaking using these pins is described in "The RTS and CTS Pins" on page 12-9.

If `FlowControl` is `hardware`, the RTS and CTS pins are automatically managed by the DTE and DCE. You can return the CTS pin value with the `PinStatus` property. Configure or return the RTS pin value with the `RequestToSend` property.

Note Some devices also use the DTR and DSR pins for handshaking. However, these pins are typically used to indicate that the system is ready for communication, and are not used to control data transmission. In MATLAB, hardware handshaking always uses the RTS and CTS pins.

If your device does not use hardware handshaking in the standard way, then you might need to manually configure the `RequestToSend` property. In this case, you should configure `FlowControl` to `none`. If `FlowControl` is `hardware`, then the `RequestToSend` value that you specify might not be honored. Refer to the device documentation to determine its specific pin behavior.

Software Handshaking

Software handshaking uses specific ASCII characters to control data flow. These characters, known as Xon and Xoff (or XON and XOFF), are described in the following table.

Software Handshaking Characters

Character	Integer Value	Description
Xon	17	Resume data transmission
Xoff	19	Pause data transmission

When using software handshaking, the control characters are sent over the transmission line the same way as regular data. Therefore, only the TD, RD, and GND pins are needed.

The main disadvantage of software handshaking is that Xon or Xoff characters are not writable while numerical data is being written to the device. This is because numerical data might contain a 17 or 19, which makes it impossible to distinguish between the control characters and the data. However, you can write Xon or Xoff while data is being asynchronously read from the device because you are using both the TD and RD pins.

Example: Using Software Handshaking

Suppose you want to use software flow control with the example described in "Example — Reading Binary Data" on page 12-47. To do this, you must configure the oscilloscope and serial port object for software flow control.

```
fprintf(s,'RS232:SOFTF ON')
s.FlowControl = 'software';
```

To pause data transfer, write the numerical value 19 to the device.

```
fwrite(s,19)
```

To resume data transfer, write the numerical value 17 to the device.

```
fwrite(s,17)
```

Debugging: Recording Information to Disk

In this section...

Introduction

Recording information to disk provides a permanent record of your serial port session, and is an easy way to debug your application. While the serial port object is connected to the device, you can record the following information to a disk file:

- The number of values written to the device, the number of values read from the device, and the data type of the values
- Data written to the device, and data read from the device
- Event information

Recording Properties

You record information to a disk file with the record function. The following table shows the properties associated with recording information to disk.

Recording Properties

Property Name	Description
RecordDetail	Amount of information saved to a record file
RecordMode	Specify whether data and event information is saved to one record file or to multiple record files

Recording Properties (Continued)

Property Name	Description
RecordName	Name of the record file
RecordStatus	Indicate if data and event information are saved to a record file

Example: Introduction to Recording Information

This example records the number of values written to and read from the device, and stores the information to the file myfile.txt.

```
s = serial('COM1');
fopen(s)
s.RecordName = 'myfile.txt';
record(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fprintf(s,'RS232?')
rs232 = fscanf(s);
```

End the serial port session.

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

You can use the type command to display myfile.txt at the command line.

Creating Multiple Record Files

When you initiate recording with the record function, the RecordMode property determines if a new record file is created or if new information is appended to an existing record file.

You can configure RecordMode to overwrite, append, or index. If RecordMode is overwrite, the record file is overwritten each time recording is initiated. If RecordMode is append, the new information is appended to the file specified by RecordName. If RecordMode is index, a different disk file is created each

time recording is initiated. The rules for specifying a record filename are discussed in the next section.

Specifying a Filename

You specify the name of the record file with the `RecordName` property. You can specify any value for `RecordName` — including a directory path — provided the filename is supported by your operating system. Additionally, if `RecordMode` is `index`, the filename follows these rules:

- Indexed filenames are identified by a number. This number precedes the filename extension and is increased by 1 for successive record files.

- If no number is specified as part of the initial filename, the first record file does not have a number associated with it. For example, if `RecordName` is `myfile.txt`, `myfile.txt` is the name of the first record file, `myfile01.txt` is the name of the second record file, and so on.

- `RecordName` is updated after the record file is closed.

- If the specified filename already exists, the existing file is overwritten.

The Record File Format

The record file is an ASCII file that contains a record of one or more serial port sessions. You specify the amount of information saved to a record file with the `RecordDetail` property.

`RecordDetail` can be `compact` or `verbose`. A compact record file contains the number of values written to the device, the number of values read from the device, the data type of the values, and event information. A verbose record file contains the preceding information as well as the data transferred to and from the device.

Binary data with precision given by `uchar`, `schar`, `(u)int8`, `(u)int16`, or `(u)int32` is recorded using hexadecimal format. For example, if the integer value 255 is read from the instrument as a 16-bit integer, the hexadecimal value 00FF is saved in the record file. Single- and double-precision floating-point numbers are recorded as decimal values using the `%g` format, and as hexadecimal values using the format specified by the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.

The IEEE floating-point format includes three components: the sign bit, the exponent field, and the significant field. Single-precision floating-point values consist of 32 bits. The value is given by

$$\text{value} = (-1)^{\text{sign}}(2^{\text{exp-127}})(1.\text{significand})$$

Double-precision floating-point values consist of 64 bits. The value is given by

$$\text{value} = (-1)^{\text{sign}}(2^{\text{exp-1023}})(1.\text{significand})$$

The floating-point format component, and the associated single-precision and double-precision bits are shown in the following table.

Component	Single-Precision Bits	Double-Precision Bits
sign	1	1
exp	2–9	2–12
significand	10–32	13–64

Bit 1 is the left-most bit as stored in the record file.

Example: Recording Information to Disk

This example illustrates how to record information transferred between a serial port object and a Tektronix TDS 210 oscilloscope. Additionally, the structure of the resulting record file is presented.

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

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

2 Connect to the device — Connect s to the oscilloscope. Because the default value for the ReadAsyncMode property is continuous, data is asynchronously returned the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

3 Configure property values — Configure s to record information to multiple disk files using the verbose format. Recording is then initiated with the first disk file defined as `WaveForm1.txt`.

```
s.RecordMode = 'index';
s.RecordDetail = 'verbose';
s.RecordName = 'WaveForm1.txt';
record(s)
```

4 Write and read data — The commands written to the instrument, and the data read from the instrument are recorded in the record file. For an explanation of the oscilloscope commands, see "Example — Writing and Reading Text Data" on page 12-44.

```
fprintf(s,'*IDN?')
idn = fscanf(s);
fprintf(s,'MEASUREMENT:IMMED:SOURCE CH2')
fprintf(s,'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(s);
```

Read the peak-to-peak voltage with the `fread` function. Note that the data returned by `fread` is recorded using hex format.

```
fprintf(s,'MEASUREMENT:MEAS1:TYPE PK2PK')
fprintf(s,'MEASUREMENT:MEAS1:VALUE?')
ptop = fread(s,s.BytesAvailable);
```

Convert the peak-to-peak voltage to a character array.

```
char(ptop)'
ans =
2.0199999809E0
```

The recording state is toggled from on to off. Because the `RecordMode` value is index, the record filename is automatically updated.

```
record(s)
s.RecordStatus
ans =
off
s.RecordName
ans =
```

```
        WaveForm2.txt
```

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

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

The Record File Contents

The contents of the WaveForm1.txt record file are shown below. Because
the RecordDetail property was verbose, the number of values, commands,
and data were recorded. Note that data returned by the fread function is in
hex format.

```
type WaveForm1.txt

Legend:
  * - An event occurred.
  > - A write operation occurred.
  < - A read operation occurred.
1      Recording on 22-Jan-2000 at 11:21:21.575. Binary data in...
2    > 6 ascii values.
       *IDN?
3    < 56 ascii values.
       TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
4    > 29 ascii values.
       MEASUREMENT:IMMED:SOURCE CH2
5    > 26 ascii values.
       MEASUREMENT:IMMED:SOURCE?
6    < 4 ascii values.
       CH2
7    > 27 ascii values.
       MEASUREMENT:MEAS1:TYPE PK2PK
8    > 25 ascii values.
       MEASUREMENT:MEAS1:VALUE?
9    < 15 uchar values.
       32 2e 30 31 39 39 39 39 39 38 30 39 45 30 0a
10     Recording off.
```

Saving and Loading

In this section...
"Using save and load" on page 12-71
"Using Serial Port Objects on Different Platforms" on page 12-72

Using save and load

You can save serial port objects to a MAT-file, just as you would any workspace variable, using the save command. For example, suppose you create the serial port object s associated with the serial port COM1, configure several property values, and perform a write and read operation.

```
s = serial('COM1');
s.BaudRate = 19200;
s.Tag = 'My serial object';
fopen(s)
fprintf(s, '*IDN?')
out = fscanf(s);
```

To save the serial port object and the data read from the device to the MAT-file myserial.mat:

```
save myserial s out
```

Note You can save data and event information as text to a disk file with the record function.

You can recreate s and out in the workspace using the load command.

```
load myserial
```

Values for read only properties are restored to their default values upon loading. For example, the Status property is restored to closed. Therefore, to use s, you must connect it to the device with the fopen function. To determine if a property is read only, examine its reference pages.

Using Serial Port Objects on Different Platforms

If you save a serial port object from one platform, and then load that object on a different platform having different serial port names, you need to modify the `Port` property value. For example, suppose you create the serial port object `s` associated with the serial port COM1 on a Microsoft Windows platform. If you want to save `s` for eventual use on a Linux platform, configure `Port` to an appropriate value such as `ttyS0` after the object is loaded.

Disconnecting and Cleaning Up

Disconnecting a Serial Port Object

When you no longer need to communicate with the device, disconnect it from the serial port object with the `fclose` function.

```
fclose(s)
```

Examine the `Status` property to verify that the serial port object and the device are disconnected.

```
s.Status
ans =
closed
```

After `fclose` is issued, the serial port associated with `s` is available. Now connect another serial port object to it using `fopen`.

Cleaning Up the MATLAB Environment

When the serial port object is no longer needed, remove it from memory with the `delete` function.

```
delete(s)
```

Before using `delete`, disconnect the serial port object from the device with the `fclose` function.

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

```
clear s
```

Use `clear` on a serial port object that is still connected to a device to remove the object from the workspace but leave it connected to the device. Restore cleared objects to MATLAB with the `instrfind` function.

Property Reference

The Property Reference Page Format

Each serial port property description contains some or all of this information:

- The property name

- A description of the property

- The property characteristics, including:

  - Read only — The condition under which the property is read only

    A property can be read-only always, never, while the serial port object is open, or while the serial port object is recording. You can configure a property value using the `set` function or dot notation. You can return the current property value using the `get` function or dot notation.

  - Data type — the property data type

    This is the data type you use when specifying a property value.

- Valid property values including the default value

  When property values are given by a predefined list, the default value is usually indicated by {}.

- An example using the property

- Related properties and functions

Serial Port Object Properties

The serial port object properties are briefly described below, and organized into categories based on how they are used. Following this section the properties are listed alphabetically and described in detail.

Communications Properties	
BaudRate	Rate at which bits are transmitted
DataBits	Number of data bits to transmit
Parity	Type of parity checking
StopBits	Number of bits used to indicate the end of a byte
Terminator	Terminator character

Write Properties	
BytesToOutput	Number of bytes currently in the output buffer
OutputBufferSize	Size of the output buffer in bytes
Timeout	Waiting time to complete a read or write operation
TransferStatus	Indicate if an asynchronous read or write operation is in progress
ValuesSent	Total number of values written to the device

Read Properties	
BytesAvailable	Number of bytes available in the input buffer
InputBufferSize	Size of the input buffer in bytes
ReadAsyncMode	Specify whether an asynchronous read operation is continuous or manual
Timeout	Waiting time to complete a read or write operation
TransferStatus	Indicate if an asynchronous read or write operation is in progress
ValuesReceived	Total number of values read from the device

Callback Properties	
BreakInterruptFcn	M-file callback function to execute when a break-interrupt event occurs

Callback Properties	
BytesAvailableFcn	M-file callback function to execute when a specified number of bytes is available in the input buffer, or a terminator is read
BytesAvailableFcnCount	Number of bytes that must be available in the input buffer to generate a bytes-available event
BytesAvailableFcnMode	Specify if the bytes-available event is generated after a specified number of bytes is available in the input buffer, or after a terminator is read
ErrorFcn	M-file callback function to execute when an error event occurs
OutputEmptyFcn	M-file callback function to execute when the output buffer is empty
PinStatusFcn	M-file callback function to execute when the CD, CTS, DSR, or RI pins change state
TimerFcn	M-file callback function to execute when a predefined period of time passes
TimerPeriod	Period of time between timer events

Control Pin Properties	
DataTerminalReady	State of the DTR pin
FlowControl	Data flow control method to use
PinStatus	State of the CD, CTS, DSR, and RI pins
RequestToSend	State of the RTS pin

Recording Properties	
RecordDetail	Amount of information saved to a record file
RecordMode	Specify whether data and event information are saved to one record file or to multiple record files

Recording Properties	
RecordName	Name of the record file
RecordStatus	Indicate if data and event information are saved to a record file

General Purpose Properties	
ByteOrder	Order in which the device stores bytes
Name	Descriptive name for the serial port object
Port	Platform-specific serial port name
Status	Indicate if the serial port object is connected to the device
Tag	Label to associate with a serial port object
Type	Object type
UserData	Data you want to associate with a serial port object

Properties — Alphabetical List

BaudRate

Purpose　　Specify the rate at which bits are transmitted

Description　　You configure BaudRate as bits per second. The transferred bits include the start bit, the data bits, the parity bit (if used), and the stop bits. However, only the data bits are stored.

The baud rate is the rate at which information is transferred in a communication channel. In the serial port context, "9600 baud" means that the serial port is capable of transferring a maximum of 9600 bits per second. If the information unit is one baud (one bit), the bit rate and the baud rate are identical. If one baud is given as 10 bits, (for example, eight data bits plus two framing bits), the bit rate is still 9600 but the baud rate is 9600/10, or 960. You always configure BaudRate as bits per second. Therefore, in the previous example, set BaudRate to 9600.

Note Both the computer and the peripheral device must be configured to the same baud rate before you can successfully read or write data.

Standard baud rates include 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 128000 and 256000 bits per second. To display the supported baud rates for the serial ports on your platform, see "Finding Serial Port Information for Your Platform" on page 12-15.

Characteristics

Read only	Never
Data type	Double

Values　　The default value is 9600.

See Also　　**Properties**

DataBits, Parity, StopBits

Purpose   Specify the M-file callback function to execute when a break-interrupt event occurs

Description   You configure `BreakInterruptFcn` to execute an M-file callback function when a break-interrupt event occurs. A break-interrupt event is generated by the serial port when the received data is in an off (space) state longer than the transmission time for one byte.

Note A break-interrupt event can be generated at any time during the serial port session.

If the `RecordStatus` property value is `on`, and a break-interrupt event occurs, the record file records this information:

- The event type as `BreakInterrupt`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

To learn how to create a callback function, see "Creating and Executing Callback Functions" on page 12-56.

Characteristics

Read only	Never
Data type	Callback function

Values   The default value is an empty string.

See Also   **Functions**

record

Properties

RecordStatus

ByteOrder

Purpose   Specify the byte order of the device

Description   You configure ByteOrder to be littleEndian or bigEndian. If
ByteOrder is littleEndian, the device stores the first byte in the first
memory address. If ByteOrder is bigEndian, the device stores the last
byte in the first memory address.

For example, suppose the hexadecimal value 4F52 is to be stored in
device memory. Because this value consists of two bytes, 4F and 52, two
memory locations are used. Using big-endian format, 4F is stored first
in the lower storage address. Using little-endian format, 52 is stored
first in the lower storage address.

Note You should configure ByteOrder to the appropriate value for your
device before performing a read or write operation. Refer to your device
documentation for information about the order in which it stores bytes.

Characteristics

Read only	Never
Data type	String

Values

{littleEndian}	The byte order of the device is little-endian.
bigEndian	The byte order of the device is big-endian.

See Also   **Properties**

Status

Purpose   Number of bytes available in the input buffer

Description   BytesAvailable indicates the number of bytes currently available to be read from the input buffer. The property value is continuously updated as the input buffer is filled, and is set to 0 after the fopen function is issued.

You can make use of BytesAvailable only when reading data asynchronously. This is because when reading data synchronously, control is returned to the MATLAB command line only after the input buffer is empty. Therefore, the BytesAvailable value is always 0. To learn how to read data asynchronously, see "Reading Text Data" on page 12-41.

The BytesAvailable value can range from zero to the size of the input buffer. Use the InputBufferSize property to specify the size of the input buffer. Use the ValuesReceived property to return the total number of values read.

Characteristics

Read only	Always
Data type	Double

Values   The default value is 0.

See Also   **Functions**

fopen

Properties

InputBufferSize, TransferStatus, ValuesReceived

BytesAvailableFcn

Purpose　　Specify the M-file callback function to execute when a specified number of bytes is available in the input buffer, or a terminator is read

Description　　You configure `BytesAvailableFcn` to execute an M-file callback function when a bytes-available event occurs. A bytes-available event occurs when the number of bytes specified by the `BytesAvailableFcnCount` property is available in the input buffer, or after a terminator is read, as determined by the `BytesAvailableFcnMode` property.

> **Note** A bytes-available event can be generated only for asynchronous read operations.

If the `RecordStatus` property value is `on`, and a bytes-available event occurs, the record file records this information:

- The event type as `BytesAvailable`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

To learn how to create a callback function, see "Creating and Executing Callback Functions" on page 12-56.

Characteristics

Read only	Never
Data type	Callback function

Values　　The default value is an empty string.

Example　　Create the serial port object `s` for a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1.

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

Configure s to execute the M-file callback function `instrcallback` when 40 bytes are available in the input buffer.

```
s.BytesAvailableFcnCount = 40;
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcn = @instrcallback;
```

Connect s to the oscilloscope.

```
fopen(s)
```

Write the `*IDN?` command, which instructs the scope to return identification information. Because the default value for the `ReadAsyncMode` property is `continuous`, data is read as soon as it is available from the instrument.

```
fprintf(s,'*IDN?')
```

MATLAB displays:

```
BytesAvailable event occurred at 18:33:35 for the object:
Serial-COM1.
```

56 bytes are read and `instrcallback` is called once. The resulting display is shown above.

```
s.BytesAvailable
ans =
    56
```

Suppose you remove 25 bytes from the input buffer and then issue the `MEASUREMENT?` command, which instructs the scope to return its measurement settings.

```
out = fscanf(s,'%c',25);
fprintf(s,'MEASUREMENT?')
```

MATLAB displays:

```
BytesAvailable event occurred at 18:33:48 for the object:
```

BytesAvailableFcn

```
Serial-COM1.

BytesAvailable event occurred at 18:33:48 for the object:
Serial-COM1.
```

There are now 102 bytes in the input buffer, 31 of which are left over from the *IDN? command. instrcallback is called twice—once when 40 bytes are available and once when 80 bytes are available.

```
s.BytesAvailable
ans =
   102
```

See Also

Functions

record

Properties

BytesAvailableFcnCount, BytesAvailableFcnMode, RecordStatus, Terminator, TransferStatus

Purpose   Specify the number of bytes that must be available in the input buffer to generate a bytes-available event

Description   You configure `BytesAvailableFcnCount` to the number of bytes that must be available in the input buffer before a bytes-available event is generated.

Use the `BytesAvailableFcnMode` property to specify whether the bytes-available event occurs after a certain number of bytes are available or after a terminator is read.

The bytes-available event executes the M-file callback function specified for the `BytesAvailableFcn` property.

You can configure `BytesAvailableFcnCount` only when the object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Read only	While open
Data type	Double

Values   The default value is `48`.

See Also   **Functions**

`fclose`

Properties

`BytesAvailableFcn`, `BytesAvailableFcnMode`, `Status`

BytesAvailableFcnMode

Purpose　　Specify if the bytes-available event is generated after a specified number of bytes is available in the input buffer, or after a terminator is read

Description　　You can configure BytesAvailableFcnMode to be terminator or byte. If BytesAvailableFcnMode is terminator, a bytes-available event occurs when the terminator specified by the Terminator property is reached. If BytesAvailableFcnMode is byte, a bytes-available event occurs when the number of bytes specified by the BytesAvailableFcnCount property is available.

The bytes-available event executes the M-file callback function specified for the BytesAvailableFcn property.

You can configure BytesAvailableFcnMode only when the object is disconnected from the device. You disconnect an object with the fclose function. A disconnected object has a Status property value of closed.

Characteristics

Read only	While open
Data type	String

Values

{terminator}	A bytes-available event is generated when the terminator is read.
byte	A bytes-available event is generated when the specified number of bytes are available.

See Also　　**Functions**

fclose

Properties

BytesAvailableFcn, BytesAvailableFcnCount, Status, Terminator

Purpose   Number of bytes currently in the output buffer

Description   `BytesToOutput` indicates the number of bytes currently in the output buffer waiting to be written to the device. The property value is continuously updated as the output buffer is filled and emptied, and is set to `0` after the `fopen` function is issued.

You can make use of `BytesToOutput` only when writing data asynchronously. This is because when writing data synchronously, control is returned to the MATLAB command line only after the output buffer is empty. Therefore, the `BytesToOutput` value is always `0`. To learn how to write data asynchronously, see "Writing Text Data" on page 12-36.

Use the `ValuesSent` property to return the total number of values written to the device.

Note If you attempt to write out more data than can fit in the output buffer, an error is returned and `BytesToOutput` is `0`. Specify the size of the output buffer with the `OutputBufferSize` property.

Characteristics

Read only	Always
Data type	Double

Values   The default value is `0`.

See Also   **Functions**

`fopen`

Properties

`OutputBufferSize`, `TransferStatus`, `ValuesSent`

DataBits

Purpose   Number of data bits to transmit

Description You can configure DataBits to be 5, 6, 7, or 8. Data is transmitted as a
series of five, six, seven, or eight bits with the least significant bit sent
first. At least seven data bits are required to transmit ASCII characters.
Eight bits are required to transmit binary data. Five and six bit data
formats are used for specialized communications equipment.

Note Both the computer and the peripheral device must be configured
to transmit the same number of data bits.

In addition to the data bits, the serial data format consists of a start bit,
one or two stop bits, and possibly a parity bit. You specify the number
of stop bits with the StopBits property, and the type of parity checking
with the Parity property.

To display the supported number of data bits for the serial ports on
your platform, see "Finding Serial Port Information for Your Platform"
on page 12-15.

Characteristics

Read only	Never
Data type	Double

Values   DataBits can be 5, 6, 7, or 8. The default value is 8.

See Also   **Properties**

Parity, StopBits

Purpose   State of the DTR pin

Description   You can configure DataTerminalReady to be on or off. If
DataTerminalReady is on, the Data Terminal Ready (DTR) pin is
asserted. If DataTerminalReady is off, the DTR pin is unasserted.

In normal usage, the DTR and Data Set Ready (DSR) pins work
together, and are used to signal if devices are connected and powered.
However, there is nothing in the RS-232 standard that states the
DTR pin must be used in any specific way. For example, DTR and
DSR might be used for handshaking. You should refer to your device
documentation to determine its specific pin behavior.

You can return the value of the DSR pin with the PinStatus
property. Handshaking is described in "Controlling the Flow of Data:
Handshaking" on page 12-62.

Characteristics

Read only	Never
Data type	String

Values

{on}	The DTR pin is asserted.
off	The DTR pin is unasserted.

See Also   **Properties**

FlowControl, PinStatus

ErrorFcn

Purpose　　Specify the M-file callback function to execute when an error event occurs

Description　　You configure ErrorFcn to execute an M-file callback function when an error event occurs.

> **Note** An error event is generated only for asynchronous read and write operations.

An error event is generated when a time-out occurs. A time-out occurs if a read or write operation does not successfully complete within the time specified by the Timeout property. An error event is not generated for configuration errors such as setting an invalid property value.

If the RecordStatus property value is on, and an error event occurs, the record file records this information:

- The event type as Error
- The error message
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

To learn how to create a callback function, see "Creating and Executing Callback Functions" on page 12-56.

Characteristics

Read only	Never
Data type	Callback function

Values　　The default value is an empty string.

See Also　　**Functions**

record

Properties

`RecordStatus, Timeout`

FlowControl

Purpose   Data flow control method to use

Description   You can configure FlowControl to be none, hardware, or software. If
FlowControl is none, data flow control (handshaking) is not used. If
FlowControl is hardware, hardware handshaking is used to control
data flow. If FlowControl is software, software handshaking is used
to control data flow.

Hardware handshaking typically utilizes the Request to Send (RTS) and
Clear to Send (CTS) pins to control data flow. Software handshaking
uses control characters (Xon and Xoff) to control data flow. For more
information about handshaking, see "Controlling the Flow of Data:
Handshaking" on page 12-62.

You can return the value of the CTS pin with the PinStatus property.
You can specify the value of the RTS pin with the RequestToSend
property. However, if FlowControl is hardware, and you specify a value
for RequestToSend, that value might not be honored.

Note Although you might be able to configure your device for both
hardware handshaking and software handshaking at the same time,
MATLAB software does not support this behavior.

Characteristics

Read only	Never
Data type	String

Values

{none}	No flow control is used.
hardware	Hardware flow control is used.
software	Software flow control is used.

See Also   **Properties**

PinStatus, RequestToSend

InputBufferSize

Purpose   Size of the input buffer in bytes

Description   You configure InputBufferSize as the total number of bytes that can
                 be stored in the input buffer during a read operation.

                 A read operation is terminated if the amount of data stored in the input
                 buffer equals the InputBufferSize value. You can read text data with
                 the fgetl, fgets, or fscanf functions. You can read binary data with
                 the fread function.

                 You can configure InputBufferSize only when the serial port object is
                 disconnected from the device. You disconnect an object with the fclose
                 function. A disconnected object has a Status property value of closed.

                 If you configure InputBufferSize while there is data in the input
                 buffer, that data is flushed.

Characteristics

Read only	While open
Data type	Double

Values   The default value is 512.

See Also   **Functions**

                 fclose, fgetl, fgets, fopen, fread, fscanf

                 Properties

                 Status

Purpose   Descriptive name for the serial port object

Description   You configure Name to be a descriptive name for the serial port object.

When you create a serial port object, a descriptive name is automatically generated and stored in Name. This name is given by concatenating the word "Serial" with the serial port specified in the serial function. However, you can change the value of Name at any time.

The serial port is given by the Port property. If you modify this property value, then Name is automatically updated to reflect that change.

Characteristics

Read only	Never
Data type	String

Values   Name is automatically defined when the serial port object is created.

Example   Suppose you create a serial port object associated with the serial port COM1.

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

s is automatically assigned a descriptive name.

```
s.Name
ans =
Serial-COM1
```

See Also   **Functions**

serial

ObjectVisibility

| **Purpose** | Control access to serial port object |

Description   The `ObjectVisibility` property provides a way for application developers to prevent end-user access to the serial port objects created by their applications. When an object's `ObjectVisibility` property is set to `off`, `instrfind` does not return or delete that object.

Objects that are not visible are still valid. If you have access to the object (for example, from within the M-file that creates it), you can set and get its properties and pass it to any function that operates on serial port objects.

Characteristics

Usage	Any serial port object
Read only	Never
Data type	String

Values

{on}	Object is visible to `instrfind`.
off	Object is not visible from the command line (except by `instrfindall`).

Examples   The following statement creates a serial port object with its `ObjectVisibility` property set to `off`:

```
s = serial('COM1','ObjectVisibility','off');
instrfind
ans =
     []
```

However, because the hidden object is in the workspace (`s`), you can access it.

```
get(s,'ObjectVisibility')
ans =
 off
```

See Also   **Functions**

instrfind, instrfindall

OutputBufferSize

Purpose   Size of the output buffer in bytes

Description You configure OutputBufferSize as the total number of bytes that can be stored in the output buffer during a write operation.

An error occurs if the output buffer cannot hold all the data to be written. You write text data with the fprintf function. You write binary data with the fwrite function.

You can configure OutputBufferSize only when the serial port object is disconnected from the device. You disconnect an object with the fclose function. A disconnected object has a Status property value of closed.

Characteristics

Read only	While open
Data type	Double

Values   The default value is 512.

See Also **Functions**

fprintf, fwrite

Properties

Status

Purpose   Specify the M-file callback function to execute when the output buffer is empty

Description   You configure OutputEmptyFcn to execute an M-file callback function when an output-empty event occurs. An output-empty event is generated when the last byte is sent from the output buffer to the device.

Note An output-empty event can be generated only for asynchronous write operations.

If the RecordStatus property value is on, and an output-empty event occurs, the record file records this information:

• The event type as OutputEmpty

• The time the event occurred using the format day-month-year hour:minute:second:millisecond

To learn how to create a callback function, see "Creating and Executing Callback Functions" on page 12-56.

Characteristics

Read only	Never
Data type	Callback function

Values   The default value is an empty string.

See Also   **Functions**

record

Properties

RecordStatus

Parity

Purpose   Specify the type of parity checking

Description   You can configure Parity to be none, odd, even, mark, or space. If
                     Parity is none, parity checking is not performed and the parity bit is
                     not transmitted. If Parity is odd, the number of mark bits (1s) in the
                     data is counted, and the parity bit is asserted or unasserted to obtain an
                     odd number of mark bits. If Parity is even, the number of mark bits
                     in the data is counted, and the parity bit is asserted or unasserted to
                     obtain an even number of mark bits. If Parity is mark, the parity bit is
                     asserted. If Parity is space, the parity bit is unasserted.

                     Parity checking can detect errors of one bit only. An error in two bits
                     might cause the data to have a seemingly valid parity, when in fact
                     it is incorrect. For more information about parity checking, see "The
                     Parity Bit" on page 12-13.

                     In addition to the parity bit, the serial data format consists of a start
                     bit, between five and eight data bits, and one or two stop bits. You
                     specify the number of data bits with the DataBits property, and the
                     number of stop bits with the StopBits property.

Characteristics

Read only	Never
Data type	String

Values

{none}	No parity checking
odd	Odd parity checking
even	Even parity checking
mark	Mark parity checking
space	Space parity checking

See Also   **Properties**

                     DataBits, StopBits

Purpose   State of the CD, CTS, DSR, and RI pins

Description   PinStatus is a structure array that contains the fields CarrierDetect, ClearToSend, DataSetReady and RingIndicator. These fields indicate the state of the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) and Ring Indicator (RI) pins, respectively. For more information about these pins, see "Serial Port Signals and Pin Assignments" on page 12-6.

PinStatus can be on or off for any of these fields. A value of on indicates the associated pin is asserted. A value of off indicates the associated pin is unasserted. A pin status event occurs when any of these pins changes its state. A pin status event executes the M-file specified by PinStatusFcn.

In normal usage, the Data Terminal Ready (DTR) and DSR pins work together, while the Request to Send (RTS) and CTS pins work together. You can specify the state of the DTR pin with the DataTerminalReady property. You can specify the state of the RTS pin with the RequestToSend property.

For an example that uses PinStatus, see "Example — Connecting Two Modems" on page 12-60.

Characteristics

Read only	Always
Data type	Structure

Values

off	The associated pin is unasserted.
on	The associated pin is asserted.

The default value is device dependent.

See Also   **Properties**

DataTerminalReady, PinStatusFcn, RequestToSend

PinStatusFcn

Purpose	Specify the M-file callback function to execute when the CD, CTS, DSR, or RI pins change state
Description	You configure `PinStatusFcn` to execute an M-file callback function when a pin status event occurs. A pin status event occurs when the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) or Ring Indicator (RI) pin changes state. A serial port pin changes state when it is asserted or unasserted. Information about the state of these pins is recorded in the `PinStatus` property.

> **Note** A pin status event can be generated at any time during the serial port session.

If the `RecordStatus` property value is `on`, and a pin status event occurs, the record file records this information:

- The event type as `PinStatus`
- The pin that changed its state, and the pin state as either `on` or `off`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

To learn how to create a callback function, see "Creating and Executing Callback Functions" on page 12-56.

Characteristics

Read only	Never
Data type	Callback function

Values The default value is an empty string.

See Also **Functions**

record

Properties

`PinStatus`, `RecordStatus`

Port

Purpose	Specify the platform-specific serial port name

Description   You configure Port to be the name of a serial port on your platform. Port specifies the physical port associated with the object and the device.

When you create a serial port object, Port is automatically assigned the port name specified for the serial function.

You can configure Port only when the object is disconnected from the device. You disconnect an object with the fclose function. A disconnected object has a Status property value of closed.

Characteristics

Read only	While open
Data type	String

Values   The Port value is determined when the serial port object is created.

Example   Suppose you create a serial port object associated with serial port COM1.

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

The value of the Port property is COM1.

```
s.Port
ans =
COM1
```

See Also   **Functions**

fclose, serial

Properties

Name, Status

Purpose
Specify whether an asynchronous read operation is continuous or manual

Description
You can configure ReadAsyncMode to be continuous or manual. If ReadAsyncMode is continuous, the serial port object continuously queries the device to determine if data is available to be read. If data is available, it is automatically read and stored in the input buffer. If issued, the readasync function is ignored.

If ReadAsyncMode is manual, the object does not query the device to determine if data is available to be read. Instead, you must manually issue the readasync function to perform an asynchronous read operation. Because readasync checks for the terminator, this function can be slow. To increase speed, configure ReadAsyncMode to continuous.

Note If the device is ready to transmit data, it will do so regardless of the ReadAsyncMode value. Therefore, if ReadAsyncMode is manual and a read operation is not in progress, data might be lost. To guarantee that all transmitted data is stored in the input buffer, you should configure ReadAsyncMode to continuous.

You can determine the amount of data available in the input buffer with the BytesAvailable property. For either ReadAsyncMode value, you can bring data into the MATLAB workspace with one of the synchronous read functions such as fscanf, fgetl, fgets, or fread.

Characteristics

Read only	Never
Data type	String

ReadAsyncMode

Values

{continuous}	Continuously query the device to determine if data is available to be read.
manual	Manually read data from the device using the readasync function.

See Also   **Functions**

fgetl, fgets, fread, fscanf, readasync

Properties

BytesAvailable, InputBufferSize

Purpose   Specify the amount of information saved to a record file

Description   You can configure RecordDetail to be compact or verbose. If
RecordDetail is compact, the number of values written to the device,
the number of values read from the device, the data type of the values,
and event information are saved to the record file. If RecordDetail is
verbose, the data written to the device, and the data read from the
device are also saved to the record file.

The event information saved to a record file is shown in the table, Event
Information on page 12-54. The verbose record file structure is shown
in "Example: Recording Information to Disk" on page 12-68.

Characteristics

Read only	Never
Data type	String

Values

{compact}	The number of values written to the device, the number of values read from the device, the data type of the values, and event information are saved to the record file.
verbose	The data written to the device, and the data read from the device are also saved to the record file.

See Also   **Functions**

record

Properties

RecordMode, RecordName, RecordStatus

RecordMode

Purpose

Specify whether data and event information are saved to one record file or to multiple record files

Description

You can configure RecordMode to be overwrite, append, or index. If RecordMode is overwrite, the record file is overwritten each time recording is initiated. If RecordMode is append, data is appended to the record file each time recording is initiated. If RecordMode is index, a different record file is created each time recording is initiated, each with an indexed filename.

You can configure RecordMode only when the object is not recording. You terminate recording with the record function. A object that is not recording has a RecordStatus property value of off.

You specify the record filename with the RecordName property. The indexed filename follows a prescribed set of rules. For a description of these rules, see "Specifying a Filename" on page 12-67.

Characteristics

Read only	While recording
Data type	String

Values

{overwrite}	The record file is overwritten.
append	Data is appended to an existing record file.
index	A different record file is created, each with an indexed filename.

Example

Suppose you create the serial port object s associated with the serial port COM1.

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

Specify the record filename with the RecordName property, configure RecordMode to index, and initiate recording.

```
s.RecordName = 'MyRecord.txt';
s.RecordMode = 'index';
record(s)
```

The record filename is automatically updated with an indexed filename after recording is turned off.

```
record(s,'off')
s.RecordName
ans =
MyRecord01.txt
```

Disconnect s from the peripheral device, remove s from memory, and remove s from the MATLAB workspace.

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

See Also   Functions

record

Properties

RecordDetail, RecordName, RecordStatus

RecordName

Purpose   Name of the record file

Description   You configure `RecordName` to be the name of the record file. You can
specify any value for `RecordName` - including a directory path - provided
the file name is supported by your operating system.

MATLAB software supports any file name supported by your operating
system. However, if you access the file with a MATLAB command, you
might need to specify the file name using single quotes. For example,
suppose you name the record file `My Record.txt`. To type this file at
the MATLAB command line, you must include the name in quotes.

```
type('My Record.txt')
```

You can specify whether data and event information are saved to one
disk file or to multiple disk files with the `RecordMode` property. If
`RecordMode` is `index`, the filename follows a prescribed set of rules. For
a description of these rules, see "Specifying a Filename" on page 12-67.

You can configure `RecordName` only when the object is not recording.
You terminate recording with the `record` function. An object that is not
recording has a `RecordStatus` property value of `off`.

Characteristics

Read only	While recording
Data type	String

Values   The default record filename is `record.txt`.

See Also   **Functions**

record

Properties

RecordDetail, RecordMode, RecordStatus

Purpose   Indicate if data and event information are saved to a record file

Description   You can configure RecordStatus to be off or on with the record function. If RecordStatus is off, then data and event information are not saved to a record file. If RecordStatus is on, then data and event information are saved to the record file specified by RecordName.

Use the record function to initiate or complete recording. RecordStatus is automatically configured to reflect the recording state.

For more information about recording to a disk file, see "Debugging: Recording Information to Disk" on page 12-65.

Characteristics

Read only	Always
Data type	String

Values

{off}	Data and event information are not written to a record file.
on	Data and event information are written to a record file.

See Also   **Functions**

record

Properties

RecordDetail, RecordMode, RecordName

RequestToSend

Purpose   State of the RTS pin

Description   You can configure RequestToSend to be on or off. If RequestToSend
is on, the Request to Send (RTS) pin is asserted. If RequestToSend is
off, the RTS pin is unasserted.

In normal usage, the RTS and Clear to Send (CTS) pins work together,
and are used as standard handshaking pins for data transfer. In this
case, RTS and CTS are automatically managed by the DTE and DCE.
However, there is nothing in the RS-232 standard that requires the
RTS pin must be used in any specific way. Therefore, if you manually
configure the RequestToSend value, it is probably for nonstandard
operations.

If your device does not use hardware handshaking in the standard
way, and you need to manually configure RequestToSend, configure
the FlowControl property to none. Otherwise, the RequestToSend
value that you specify might not be honored. Refer to your device
documentation to determine its specific pin behavior.

You can return the value of the CTS pin with the PinStatus
property. Handshaking is described in "Controlling the Flow of Data:
Handshaking" on page 12-62.

Characteristics

Read only	Never
Data type	String

Values

{on}	The RTS pin is asserted.
off	The RTS pin is unasserted.

See Also   **Properties**

FlowControl, PinStatus

Purpose   Indicate if the serial port object is connected to the device

Description   Status can be open or closed. If Status is closed, the serial port object is not connected to the device. If Status is open, the serial port object is connected to the device.

Before you can write or read data, you must connect the serial port object to the device with the fopen function. Use the fclose function to disconnect a serial port object from the device.

Characteristics

Read only	Always
Data type	String

Values

{closed}	The serial port object is not connected to the device.
open	The serial port object is connected to the device.

See Also   **Functions**

fclose, fopen

StopBits

Purpose   Number of bits used to indicate the end of a byte

Description   You can configure StopBits to be 1, 1.5, or 2. If StopBits is 1, one stop bit is used to indicate the end of data transmission. If StopBits is 2, two stop bits are used to indicate the end of data transmission. If StopBits is 1.5, the stop bit is transferred for 150% of the normal time used to transfer one bit.

> **Note** Both the computer and the peripheral device must be configured to transmit the same number of stop bits.

In addition to the stop bits, the serial data format consists of a start bit, between five and eight data bits, and possibly a parity bit. You specify the number of data bits with the DataBits property, and the type of parity checking with the Parity property.

Characteristics

Read only	Never
Data type	Double

Values

{1}	One stop bit is transmitted to indicate the end of a byte.
1.5	The stop bit is transferred for 150% of the normal time used to transfer one bit.
2	Two stop bits are transmitted to indicate the end of a byte.

See Also   **Properties**

DataBits, Parity

Purpose   Label to associate with a serial port object

Description   You configure Tag to be a string value that uniquely identifies a serial port object.

Tag is particularly useful when constructing programs that would otherwise need to define the serial port object as a global variable, or pass the object as an argument between callback routines.

You can return the serial port object with the instrfind function by specifying the Tag property value.

Characteristics

Read only	Never
Data type	String

Values   The default value is an empty string.

Example   Suppose you create a serial port object associated with the serial port COM1.

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

You can assign s a unique label using Tag.

```
set(s,'Tag','MySerialObj')
```

You can access s in the MATLAB workspace or in an M-file using the instrfind function and the Tag property value.

```
s1 = instrfind('Tag','MySerialObj');
```

See Also   **Functions**

instrfind

Terminator

Purpose   Terminator character

Description   You can configure Terminator to an integer value ranging from 0 to
                     127, which represents the ASCII code for the character, or you can
                     configure Terminator to the ASCII character. For example, to configure
                     Terminator to a carriage return, specify the value to be CR or 13. To
                     configure Terminator to a linefeed, specify the value to be LF or 10. You
                     can also set Terminator to CR/LF or LF/CR. If Terminator is CR/LF, the
                     terminator is a carriage return followed by a line feed. If Terminator is
                     LF/CR, the terminator is a linefeed followed by a carriage return. Note
                     that there are no integer equivalents for these two values. Additionally,
                     you can set Terminator to a 1-by-2 cell array. The first element of the
                     cell is the read terminator and the second element of the cell array is
                     the write terminator.

                     When performing a write operation using the fprintf function, all
                     occurrences of \n are replaced with the Terminator property value.
                     Note that %s\n is the default format for fprintf. A read operation with
                     fgetl, fgets, or fscanf completes when the Terminator value is read.
                     The terminator is ignored for binary operations.

                     You can also use the terminator to generate a bytes-available event
                     when the BytesAvailableFcnMode is set to terminator.

Characteristics

Read only	Never
Data type	String

Values   An integer value ranging from 0 to 127, or the equivalent ASCII
                     character. CR/LF and LF/CR are also accepted values. You specify
                     different read and write terminators as a 1-by-2 cell array.

See Also   **Functions**

                     fgetl, fgets, fprintf, fscanf

Properties

BytesAvailableFcnMode

Timeout

Purpose   Waiting time to complete a read or write operation

Description   You configure Timeout to be the maximum time (in seconds) to wait to complete a read or write operation.

If a time-out occurs, the read or write operation aborts. Additionally, if a time-out occurs during an asynchronous read or write operation, then:

- An error event is generated.
- The M-file callback function specified for ErrorFcn is executed.

Characteristics

Read only	Never
Data type	Double

Values   The default value is 10 seconds.

See Also   **Properties**

ErrorFcn

Purpose   Specify the M-file callback function to execute when a predefined period of time passes.

Description   You configure TimerFcn to execute an M-file callback function when a timer event occurs. A timer event occurs when the time specified by the TimerPeriod property passes. Time is measured relative to when the serial port object is connected to the device with fopen.

Note A timer event can be generated at any time during the serial port session.

If the RecordStatus property value is on, and a timer event occurs, the record file records this information:

- The event type as Timer
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Some timer events might not be processed if your system is significantly slowed or if the TimerPeriod value is too small.

To learn how to create a callback function, see "Creating and Executing Callback Functions" on page 12-56.

Characteristics

Read only	Never
Data type	Callback function

Values   The default value is an empty string.

See Also   **Functions**

fopen, record

TimerFcn

Properties

`RecordStatus, TimerPeriod`

Purpose   Period of time between timer events

Description   TimerPeriod specifies the time, in seconds, that must pass before the callback function specified for TimerFcn is called. Time is measured relative to when the serial port object is connected to the device with fopen.

Some timer events might not be processed if your system is significantly slowed or if the TimerPeriod value is too small.

Characteristics

Read only	Never
Data type	Callback function

Values   The default value is 1 second. The minimum value is 0.01 second.

See Also   **Functions**

fopen

Properties

TimerFcn

TransferStatus

Purpose Indicate if an asynchronous read or write operation is in progress

Description TransferStatus can be idle, read, write, or read&write. If TransferStatus is idle, no asynchronous read or write operations are in progress. If TransferStatus is read, an asynchronous read operation is in progress. If TransferStatus is write, an asynchronous write operation is in progress. If TransferStatus is read&write, both an asynchronous read and an asynchronous write operation are in progress.

You can write data asynchronously using the fprintf or fwrite functions. You can read data asynchronously using the readasync function, or by configuring the ReadAsyncMode property to continuous. While readasync is executing, TransferStatus might indicate that data is being read even though data is not filling the input buffer. If ReadAsyncMode is continuous, TransferStatus indicates that data is being read only when data is actually filling the input buffer.

You can execute an asynchronous read and an asynchronous write operation simultaneously because serial ports have separate read and write pins. For more information about synchronous and asynchronous read and write operations, see "Writing and Reading Data" on page 12-31.

Characteristics

Read only	Always
Data type	String

Values

{idle}	No asynchronous operations are in progress.
read	An asynchronous read operation is in progress.

write	An asynchronous write operation is in progress.
read&write	Asynchronous read and write operations are in progress.

See Also   **Functions**

fprintf, fwrite, readasync

Properties

ReadAsyncMode

Type

Purpose	Object type

Description Type indicates the type of the object. Type is automatically defined after the serial port object is created with the serial function. The Type value is always serial.

Characteristics

Read only	Always
Data type	String

Values Type is always serial. This value is automatically defined when the serial port object is created.

Example Suppose you create a serial port object associated with the serial port COM1.

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

The value of the Type property is serial, which is the object class.

```
s.Type
ans =
serial
```

You can also display the object class with the whos command.

```
Name      Size         Bytes  Class

  s       1x1            644  serial object

Grand total is 18 elements using 644 bytes
```

See Also **Functions**

serial

Purpose   Data you want to associate with a serial port object

Description You configure UserData to store data that you want to associate with a
serial port object. The object does not use this data directly, but you can
access it using the get function or the dot notation.

Characteristics

Read only	Never
Data type	Any type

Values   The default value is an empty vector.

Example   Suppose you create the serial port object associated with the serial port
COM1.

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

You can associate data with s by storing it in UserData.

```
coeff.a = 1.0;
coeff.b = -1.25;
s.UserData = coeff;
```

ValuesReceived

Purpose   Total number of values read from the device

Description   ValuesReceived indicates the total number of values read from the device. The value is updated after each successful read operation, and is set to 0 after the fopen function is issued. If the terminator is read from the device, then this value is reflected by ValuesReceived.

If you are reading data asynchronously, use the BytesAvailable property to return the number of bytes currently available in the input buffer.

When performing a read operation, the received data is represented by values rather than bytes. A value consists of one or more bytes. For example, one uint32 value consists of four bytes. For more information about bytes and values, see "Bytes Versus Values" on page 12-11.

Characteristics

Read only	Always
Data type	Double

Values   The default value is 0.

Example   Suppose you create a serial port object associated with the serial port COM1.

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

If you write the RS232? command, and read back the response using fscanf, ValuesReceived is 17 because the instrument is configured to send the LF terminator.

```
fprintf(s,'RS232?')
out = fscanf(s)
out =
9600;0;0;NONE;LF
s.ValuesReceived
```

```
ans =
    17
```

See Also

Functions

`fopen`

Properties

`BytesAvailable`

ValuesSent

Purpose   Total number of values written to the device

Description ValuesSent indicates the total number of values written to the device. The value is updated after each successful write operation, and is set to 0 after the fopen function is issued. If you are writing the terminator, ValuesSent reflects this value.

If you are writing data asynchronously, use the BytesToOutput property to return the number of bytes currently in the output buffer.

When performing a write operation, the transmitted data is represented by values rather than bytes. A value consists of one or more bytes. For example, one uint32 value consists of four bytes. For more information about bytes and values, see "Bytes Versus Values" on page 12-11.

Characteristics

Read only	Always
Data type	Double

Values   The default value is 0.

Example   Suppose you create a serial port object associated with the serial port COM1.

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

If you write the *IDN? command using the fprintf function, ValuesSent is 6 because the default data format is %s\n, and the terminator was written.

```
fprintf(s,'*IDN?')
s.ValuesSent
ans =
    6
```

See Also   **Functions**

fopen

Properties

BytesToOutput

ValuesSent

Examples

Use this list to find examples in the documentation.

Importing and Exporting Data

MATLAB Interface to Generic DLLs

Calling C and Fortran Programs from MATLAB

Creating C Language MEX-Files

Creating Fortran MEX-Files

Calling MATLAB from C and Fortran Programs

Calling Java from MATLAB

COM Support

Serial Port I/O