MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

The MATH WORKS Inc.

Application Program Interface Guide

Version 5

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500 Mail

`http://www.mathworks.com` Web
`ftp.mathworks.com` Anonymous FTP server
`comp.soft-sys.matlab` Newsgroup

`support@mathworks.com` Technical support
`suggest@mathworks.com` Product enhancement suggestions
`bugs@mathworks.com` Bug reports
`doc@mathworks.com` Documentation error reports
`subscribe@mathworks.com` Subscribing user registration
`service@mathworks.com` Order status, license renewals, passcodes
`info@mathworks.com` Sales, pricing, and general information

Application Program Interface Guide

Printing History:

December 1996	First printing	
July 1997	Revised for 5.1 (online only)	
January 1998	Second printing	Revised for MATLAB 5.2
October 1998	Third printing	Revised for MATLAB 5.3 (Release 11)

Contents

1 Introducing the MATLAB API

2 Getting Started

3 Creating C Language MEX-Files

4 Creating Fortran MEX-Files

5 Data Export and Import

6 Using the MATLAB Engine

7 Client/Server Applications

8 System Setup

A API Functions

Directory Organization

B

1

Introducing the MATLAB API

Introduction to MATLAB API

Although MATLAB® 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 an Application Program Interface (API) to support these external interfaces. The functions supported by the API include:

- Calling C or Fortran programs from MATLAB.
- Importing and exporting data to and from the MATLAB environment.
- Establishing client/server relationships between MATLAB and other software programs.

MEX-Files

You can call your own C or Fortran subroutines from MATLAB as if they were built-in functions. MATLAB callable C and Fortran programs are referred to as MEX-files. MEX-files are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute.

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.
- Bottleneck computations (usually `for`-loops) that do not run fast enough in MATLAB can be recoded in C or Fortran for efficiency.

This book uses many examples to show how to write C and Fortran MEX-files.

MAT-File Applications

You can use MAT-files, the data file format MATLAB uses for saving data to disk, to import data to and export data from the MATLAB environment. MAT-files provide a convenient mechanism for moving your MATLAB data between different platforms in a highly portable manner. In addition, they provide a means to import and export your data to other stand-alone MATLAB applications. To simplify your use of MAT-files in applications outside of MATLAB, we provide a library of access routines 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` API routines discussed in this book.

Engine Applications

MATLAB provides a set of routines that allows you to call MATLAB from your own programs, thereby employing MATLAB as a computation engine. MATLAB engine programs are C or Fortran programs that communicate with a separate MATLAB process via pipes (in UNIX) and through ActiveX on Windows. There is a library of functions provided with MATLAB that allows you to start and end the MATLAB process, send data to and from MATLAB, and send commands to be processed in MATLAB.

Some of the things you can do with the MATLAB engine are:

- Call a math routine 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.

MATLAB Data

The MATLAB Array

Before you can program MEX-files, you must understand how MATLAB represents the many data types it supports. 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. This 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

Data Types in MATLAB

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.

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. Each character in the string is stored as 16-bit ASCII Unicode. Unlike C, MATLAB strings are not null terminated.

Sparse Matrices

Sparse matrices have a different storage convention in MATLAB. The parameters `pr` and `pi` are still arrays of double-precision numbers, but 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 that contains column index information. For j, in the range 0 ≤ j ≤ N–1, jc[j] is the index in ir and pr (and pi if it exists) of the first nonzero entry in the jth column and jc[j+1] – 1 index of the last nonzero entry. As a result, jc[N] is also equal to nnz, the number of nonzero entries in the matrix. If nnz is less than nzmax, then more nonzero entries can be inserted in the array without allocating additional storage.

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 classname 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.

Logical Arrays

Any noncomplex numeric or sparse array can be flagged as logical. The storage for a logical array is the same as the storage for a nonlogical array.

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.

Using Data Types

The six fundamental data types in MATLAB are `double`, `char`, `sparse`, `uint8`, `cell`, and `struct`. You can write 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 C and Fortran MEX-files, once compiled, exactly like M-functions.

The explore Example

There is an example MEX-file included with MATLAB, called `explore`, that identifies the data type of an input variable. For example, typing

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

produces this result

```
------------------------------------------------
Name: x
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)
```

API Documentation

The API Documentation Set

This book, the *Application Program Interface Guide*, contains configuration information and tutorials for using the MATLAB API. The complete set of reference documentation for all the API-related functions is provided online, and can be accessed from the MATLAB Help Desk by typing `helpdesk` at the MATLAB prompt. From the Help Desk, you can also access online (PDF) versions of the *Application Program Interface Guide* and the *Application Program Interface Reference*. The online version of the *Application Program Interface Reference* is the complete set of API reference pages in a book format. If you need a printed version of the API reference pages, you can easily print the PDF version of the *Application Program Interface Reference*.

API Tutorial Files

In addition to the printed *Application Program Interface Guide* and the online *Application Program Interface Reference* that is accessible via the Help Desk, there are many sample files included with MATLAB that can help you learn how to use the API. The `mex` and `mx` subdirectories in the `extern/examples` directory contain examples that are referenced from the `mex` and `mx` functions in the online *Application Program Interface Reference*.

The `refbook` subdirectory in the `extern/examples` directory contains the MEX-file examples (C and Fortran) that are used in this book, the *Application Program Interface Guide*. The `eng_mat` subdirectory in the `extern/examples` directory contains examples that are referenced from the engine and MAT-file routines in the online *Application Program Interface Reference* and the engine (Chapter 6) and MAT-file (Chapter 5) chapters in this book.

Note: You can find the most recent versions of the example programs from this book at the anonymous FTP server:

```
ftp.mathworks.com/pub/tech-support/library/matlab5/extern/examples/
refbook
```

You can find the most recent versions of the examples described in the online *Application Program Interface Reference* at:

```
ftp.mathworks.com/pub/tech-support/library/matlab5/extern/examples/mex
ftp.mathworks.com/pub/tech-support/library/matlab5/extern/examples/mx
ftp.mathworks.com/pub/tech-support/library/matlab5/extern/examples/
eng_mat
```

How This Book Is Organized

Chapter 1 provides an overview of MEX-files, MAT-file applications, engine applications, and the way MATLAB stores its data. This chapter also describes the API documentation set.

Chapter 2 discusses MEX-files, which enable you to call your own C or Fortran subroutines directly from MATLAB. It also provides basic information to get you up and running so that you can configure your system to build MEX-functions.

Chapters 3 and 4 contain C and Fortran examples, which explain how to create MEX-files on different platforms.

Chapter 5 continues with a discussion of techniques for importing and exporting data to and from the MATLAB environment. The most important technique is MAT-files – the files MATLAB uses for saving data to a disk. MAT-files offer a simple and convenient mechanism for transporting your data between different platforms. They also enable you to import and export your MATLAB data to and from other MATLAB stand-alone applications. To simplify the use of MAT-files with other applications, a library of access routines is provided, which makes it very easy to read and write MAT-files using your own C or Fortran programs.

Chapter 6 discusses the MATLAB engine, which enables you to set up client/server relationships between MATLAB and other software programs, such as Excel.

Chapter 7 includes information on ActiveX, which is a component integration technology for Microsoft Windows. This chapter also contains information on dynamic data exchange (DDE) software that allows Microsoft Windows applications to communicate with each other by exchanging data.

Chapter 8 focuses on platform-specific issues and provides detailed information on the `mex` script. In addition, Chapter 8 contains information on troubleshooting and memory management.

The Appendices contain supplemental information regarding the MATLAB API. Appendix A lists the set of API functions including C and Fortran MX-functions, C and Fortran MEX-functions, C and Fortran MAT-file routines, C and Fortran engine routines, and DDE routines. Appendix B describes the directory organization and purpose of the files associated with the MATLAB API.

2

Getting Started

Introducing MEX-Files

MEX-files are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute. MEX-files are not appropriate for all applications. MATLAB is a high-productivity system whose specialty is eliminating time-consuming, low-level programming in compiled languages like Fortran or C. In general, most programming should be done in MATLAB. Don't use the MEX facility unless your application requires it.

Using MEX-Files

MEX-files are subroutines produced from 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. Table 2-1 lists the platform-specific extensions for MEX-files.

Table 2-1: MEX-File Extensions

Platform	MEX-File Extension
Sun OS 4.x	`mex4`
HP 9000/series 700	`mexhp7`
Alpha	`mexaxp`
SGI	`mexsg`
SGI 64	`mexsg64`
IBM RS/6000	`mexrs6`
Linux	`mexlx`
Solaris	`mexsol`
Windows	`dll`

You can call MEX-files exactly as you would call any M-function. For example, a MEX-file called `conv2.mex` on your disk in the MATLAB `datafun` toolbox directory performs a 2-D convolution of matrices. `conv2.m` only contains the help text documentation. If you invoke the function `conv2` from inside

MATLAB, the interpreter looks through the list of directories on MATLAB's search path. It scans each directory looking for the first occurrence of a file named `conv2` with the corresponding filename extension from the table or `.m`. When it finds one, it loads the file and executes it. MEX-files take precedence over M-files when like-named files exist in the same directory. However, help text documentation is still read from the `.m` file.

The Distinction Between mx and mex Prefixes

Routines in the 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.

Building MEX-Files

Your installed version of MATLAB contains all the tools you need to work with the API, except a C or Fortran compiler. Depending on your requirements, you'll need either an ANSI C compiler or a Fortran compiler. Also, if you are working on a Microsoft Windows platform, your compiler must be able to create 32-bit windows dynamically linked libraries (DLLs).

The API supports many compilers and provides options files designed specifically for these compilers. Chapter 8, "System Setup," provides detailed information on the compilers, options files, and customization. There is also additional information regarding options files later in this chapter in "Using –f to Specify an Options File."

Depending on your platform, you may have to do some preliminary work before you can create MEX-files with the `mex` script. The next section, "Testing Your Configuration," takes you through the process of creating a MEX-file on each platform.

Note: The MathWorks provides an option (`setup`) for the `mex` script that lets you easily choose or switch your compiler on Windows systems.

More detailed information about the `mex` script is provided in "Custom Building MEX-Files" in Chapter 8. In addition, Chapter 8 contains a "Troubleshooting" section if you are having difficulties creating MEX-files.

Testing Your Configuration

The quickest way to check if your system is set up properly to create MEX-files is by trying the actual process. There is C source code for an example, `yprime.c`, and its Fortran counterpart, `yprimef.f` and `yprimefg.f` (Windows) and `yprimef.F` and `yprimefg.F` (UNIX), included in the

<matlab>/extern/examples/mex directory, where <matlab> 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 <matlab>/extern/examples/mex.

The following sections contain configuration information for creating MEX-files on UNIX and Windows systems. If, after following the instructions, you have difficulty creating MEX-files, refer to Chapter 8 for additional troubleshooting information.

On UNIX

To compile and link the example source files, yprime.c or yprimef.F and yprimefg.F, on UNIX, you must first copy the file(s) to a local directory, and then change directory (cd) to that local directory.

At the MATLAB prompt, type

```
mex yprime.c
```

This should create the MEX-file called yprime with the appropriate extension for your system.

You can now call yprime as if it were an M-function.

```
yprime(1,1:4)
ans =
     2.0000    8.9685    4.0000    -1.0947
```

To try the Fortran version of the sample program with your Fortran compiler, at the MATLAB prompt, type

```
mex yprimef.F yprimefg.F
```

In addition to running the mex script from the MATLAB prompt, you can also run the script from the system prompt.

On Windows

Configuring an Options File. Before you can create MEX-files on the Windows platform, you *must* configure the default options file, `mexopts.bat`, for your compiler. The switch, `setup`, provides an easy way for you to configure the default options file. You can run the `setup` option from either the MATLAB or DOS command prompt, and it can be called anytime to configure or change the options file.

Executing the `setup` option presents a list of compilers whose options files are currently shipped in the `bin` subdirectory of MATLAB. This example shows how to select the Microsoft Visual C++ compiler.

```
mex –setup

Welcome to the utility for setting up compilers
for building external interface files.


Choose your C/C++ compiler:
[1] Borland C/C++        (version 5.0)
[2] Microsoft Visual C++ (version 4.2 or version 5.0)
[3] Watcom C/C++         (version 10.6 or version 11)

Fortran compilers
[4] DIGITAL Visual Fortran (version 5.0)

[0] None

compiler: 2
```

If the selected compiler has more than one options file (due to more than one version of the compiler), you are asked for a specific version. For example,

```
Choose the version of your C/C++ compiler:
[1] Microsoft Visual C++ 4.2
[2] Microsoft Visual C++ 5.0

version: 1
```

You are then asked to enter the root directory of your compiler installation.

```
Please enter the location of your C/C++ compiler: [c:\msdev]
```

Note: Some compilers create a directory tree under their root directory when you install them. You must respond to this prompt with the root directory only. For example, if the compiler creates directories `bin`, `lib`, and `include` under `c:\msdev`, you should enter only the root directory, which is `c:\msdev`.

Finally, you are asked to verify your choices.

```
Please verify your choices:

Compiler: Microsoft Visual C++ 4.2
Location: c:\msdev
Are these correct?([y]/n): y

Default options file is being updated...
```

Building a MEX-File. To compile and link the example source file on Windows, at the MATLAB prompt, type

```
cd([matlabroot '\extern\examples\mex'])
mex yprime.c
```

This should create the MEX-file called `yprime` with the `.DLL` extension, which corresponds to the Windows platform.

You can now call `yprime` as if it were an M-function.

```
yprime(1,1:4)
ans =
     2.0000    8.9685    4.0000    -1.0947
```

To try the Fortran version of the sample program with your Fortran compiler (`mex -setup` allows you to change compilers anytime), at the MATLAB prompt, type

```
cd([matlabroot '\extern\examples\mex'])
mex yprimef.f yprimefg.f
```

In addition to running the mex script from the MATLAB prompt, you can also run the script from the system prompt.

Using –f to Specify an Options File

Windows users can use the –setup option to specify an options file. In addition, all users (UNIX and Windows) can use the –f option to specify an options file. To use the –f option, at the MATLAB prompt type

```
mex filename –f <optionsfile>
```

and specify the name of the options file. Table 2-2 contains a list of the options files included with MATLAB.

There are several situations when it may be necessary to specify an options file every time you use the mex script. These include:

- *(Windows)* You want to use a different compiler (and not use the –setup option), or you want to compile MAT or engine stand-alone programs.
- *(UNIX)* You do not want to use the system C compiler.

Preconfigured Options Files

MATLAB includes some preconfigured options files that you can use with particular compilers. Table 2-2 lists the compilers whose options files are included with this release of MATLAB.

Table 2-2: Options Files

Platform	Compiler	Options File
Windows	Microsoft C/C++, Version 4.2 Microsoft C/C++, Version 5.0	`msvcopts.bat` `msvc50opts.bat`
	DIGITAL Visual Fortran, Version 5.0	`df50opts.bat`
	Watcom C/C++, Version 10.6 Watcom C/C++, Version 11	`watcopts.bat` `wat11copts.bat`
	Borland C++, Version 5.0	`bccopts.bat`
	Watcom C for Engine and MAT stand-alone programs, Version 10.6 Watcom C for Engine and MAT stand-alone programs, Version 11	`watengmatopts.bat` `wat11engmatopts.bat`
	Microsoft Visual C for Engine and MAT stand-alone programs, Version 4.2 Microsoft Visual C for Engine and MAT stand-alone programs, Version 5.0	`msvcengmatopts.bat` `msvc50engmatopts.bat`
	Borland C for Engine and MAT stand-alone programs, Version 5.0	`bccengmatopts.bat`
	DIGITAL Visual Fortran for MAT stand-alone programs, Version 5.0	`df50engmatopts.bat`

Table 2-2: Options Files (Continued)

Platform	Compiler	Options File
UNIX	System ANSI Compiler	`mexopts.sh`
	GCC	`gccopts.sh`
	System C++ Compiler	`cxxopts.sh`

Note: An up-to-date list of options files is available from our FTP server:

`ftp.mathworks.com/pub/tech-support/library/matlab5/bin`

For a list of all the compilers supported by MATLAB, see the MathWorks Technical Support Department's Technical Notes at

`http://www.mathworks.com/support/tech-notes/#mex`

Table 2-3 shows where the default options files are located on each platform.

Table 2-3: Options Files Path

Platform	Location
UNIX	`<matlab>/bin`
Windows	`<matlab>\bin`

If you want to use one of these options files,

1. Copy the desired options file to the directory where you are creating your MEX-files. Options files are not M-files, so they do not automatically appear in the MATLAB path.

2. Specify the `–f <optionsfile>` switch in the `mex` command using the filename of the desired options file.

Alternatively, you do not have to copy the options file to the MEX-file creation directory; you can specify the options filename, including the full path, in the options filename.

Note: Chapter 8 contains specific information on how to modify options files for particular systems.

Troubleshooting Your Configuration

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

Search Path Problem on Windows

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

MATLAB Pathnames Containing Spaces on Windows

If you have problems building MEX-files on Windows and there is a space in any of the directory names within the MATLAB path, you need to either reinstall MATLAB into a pathname 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. Also, if you install MATLAB in a directory such as `MATLAB V5.2`, you may have difficulty.

DLLs Not on Path on Windows

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

Non-ANSI Compiler on UNIX

On the Sun OS 4.1.* platform, the bundled compiler is not ANSI; you must acquire a supported ANSI compiler. The same is true on the HP-700; you must acquire a supported ANSI compiler.

General Configuration Problem

Make sure you followed the configuration steps for your platform described in this chapter. Also, refer to Chapter 8, “System Setup,” for additional information.

3

Creating C Language MEX-Files

C MEX-Files

C MEX-files are built by using the `mex` script to compile your C source code with additional calls to API routines.

Directory Organization

A collection of files associated with the creation of C language MEX-files is located on your disk. This table lists the location of these files.

Platform	Directory
Windows	`<matlab>\extern`
UNIX	`<matlab>/extern`
	where: `<matlab>` is the MATLAB root directory

Appendix B, "Directory Organization," describes the contents of the API-related directories and files.

The Parts of a MEX-File

The source code for a MEX-file consists of two distinct parts:

- A *computational routine* that contains the code for performing the computations that you want implemented in the MEX-file. Computations can be numerical computations as well as inputting and outputting data.
- A *gateway routine* that interfaces the computational routine with MATLAB by the entry point `mexFunction` and its parameters `prhs`, `nrhs`, `plhs`, `nlhs`, where `prhs` is an array of right-hand input arguments, `nrhs` is the number of right-hand input arguments, `plhs` is an array of left-hand output arguments, and `nlhs` is the number of left-hand output arguments. The gateway calls the computational routine as a subroutine.

In the gateway routine, you can access the data in the `mxArray` structure and then manipulate this data in your C 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. After calling your C computational

routine from the gateway, you can set a pointer of type `mxArray` to the data it returns. MATLAB is then able to recognize the output from your computational routine as the output from the MEX-file.

Figure 3-1 shows how inputs enter a MEX-file, what functions the gateway function performs, and how outputs return to MATLAB.

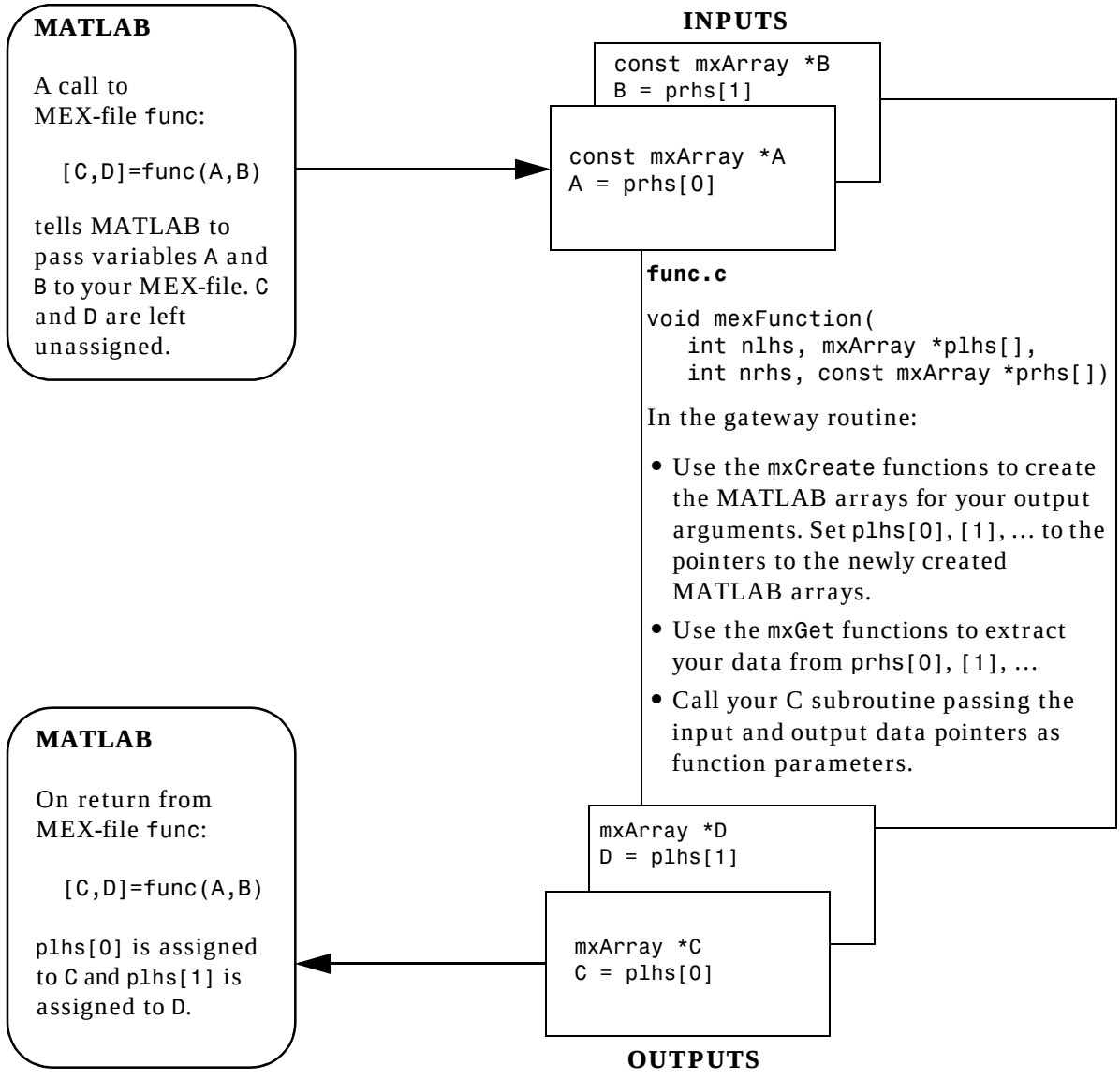


Figure 3-1: C MEX Cycle

Required Arguments to a MEX-File

The two components of the MEX-file may be separate or combined. In either case, the files must contain the `#include "mex.h"` header so that the entry point and interface routines are declared properly. The name of the gateway routine must always be `mexFunction` and must contain these parameters:

```
void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
   /* more C code ... */
```

The parameters `nlhs` and `nrhs` contain the number of left- and right-hand arguments with which the MEX-file is invoked. In the syntax of the MATLAB language, functions have the general form

```
[a,b,c,…] = fun(d,e,f,…)
```

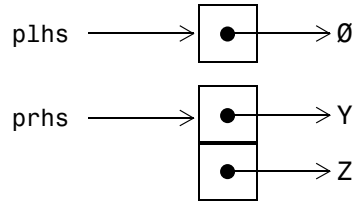
where the ellipsis (…) denotes additional terms of the same format. The `a,b,c,…` are left-hand arguments and the `d,e,f,…` are right-hand arguments.

The parameters `plhs` and `prhs` are vectors that contain pointers to the left- and right-hand arguments of the MEX-file. Note that both are declared as containing type `mxArray *`, which means that the variables pointed at are MATLAB arrays. `prhs` is a length `nrhs` array of pointers to the right-hand side inputs to the MEX-file, and `plhs` is a length `nlhs` array that will contain pointers to the left-hand side outputs that your function generates. For example, if you invoke a MEX-file from the MATLAB workspace with the command

```
x = fun(y,z);
```

the MATLAB interpreter calls `mexFunction` with the arguments

```
nlhs = 1

nrhs = 2
```



`plhs` is a 1-element C array where the single element is a `null` pointer. `prhs` is a 2-element C array where the first element is a pointer to an `mxArray` named `Y` and the second element is a pointer to an `mxArray` named `Z`.

The parameter `plhs` points at nothing because the output `x` is not created until the subroutine executes. It is the responsibility of the gateway routine to create an output array and to set a pointer to that array in `plhs[0]`. If `plhs[0]` is left unassigned, MATLAB prints a warning message stating that no output has been assigned.

Note: It is possible to return an output value even if `nlhs = 0`. This corresponds to returning the result in the `ans` variable.

Examples of C MEX-Files

The next sections of this chapter include examples of different MEX-files. The MATLAB 5 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 data types.

Note: You can find the most recent versions of the example programs from this chapter at the anonymous FTP server:

`ftp.mathworks.com/pub/tech-support/library/matlab5/extern/examples/refbook`

A First Example

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.

```
#include <math.h>
void timestwo(double y[], double x[])
{
  y[0] = 2.0*x[0];
  return;
}
```

Below is the same function written in the MEX-file format.

```
#include "mex.h"

/*
 * timestwo.c - example found in API guide
 *
 * Computational function that takes a scalar and doubles it.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-1998 The MathWorks, Inc.
 */

/* $Revision: 1.5 $ */

void timestwo(double y[], double x[])
{
  y[0] = 2.0*x[0];
}

void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
  double *x,*y;
  int     mrows,ncols;

  /* Check for proper number of arguments. */
  if(nrhs!=1) {
    mexErrMsgTxt("One input required.");
  } else if(nlhs>1) {
    mexErrMsgTxt("Too many output arguments");
  }

  /* The input must be a noncomplex scalar double.*/
  mrows = mxGetM(prhs[0]);
  ncols = mxGetN(prhs[0]);
  if( !mxIsDouble(prhs[0]) || mxIsComplex(prhs[0]) ||
      !(mrows==1 && ncols==1) ) {
    mexErrMsgTxt("Input must be a noncomplex scalar double.");
  }
```

```
/* Create matrix for the return argument. */
  plhs[0] = mxCreateDoubleMatrix(mrows,ncols, mxREAL);

  /* Assign pointers to each input and output. */
  x = mxGetPr(prhs[0]);
  y = mxGetPr(plhs[0]);

  /* Call the timestwo subroutine. */
  timestwo(y,x);
}
```

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 source file at the MATLAB prompt, type

```
mex timestwo.c
```

This carries out the necessary steps to create the 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 Windows and `mex.sh` on UNIX. 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. This is the alternative code (error checking has been omitted for brevity).

```
#include "mex.h"

/*
 * timestwoalt.c - example found in API guide
 *
 * Use mxGetScalar to return the values of scalars instead of
 * pointers to copies of scalar variables.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-1998 The MathWorks, Inc.
 */

/* $Revision: 1.2 $ */
void timestwo_alt(double *y, double x)
{
  *y = 2.0*x;
}

void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
  double *y;
  double  x;

  /* Create a 1-by-1 matrix for the return argument. */
  plhs[0] = mxCreateDoubleMatrix(1,1,mxREAL);

  /* Get the scalar value of the input x. */
  /* Note: mxGetScalar returns a value, not a pointer. */
  x = mxGetScalar(prhs[0]);

  /* Assign a pointer to the output. */
  y = mxGetPr(plhs[0]);

  /* Call the timestwo_alt subroutine. */
  timestwo_alt(y,x);
}
```

This example passes the input scalar `x` by value into the `timestwo_alt` subroutine, but passes the output scalar `y` by reference.

Manipulating Strings

Any MATLAB data type can be passed to and from MEX-files. For example, this C code accepts a string and returns the characters in reverse order.

```
/* $Revision: 1.7 $ */
/*=========================================================
 * revord.c
 * Example for illustrating how to copy the string data from
 * MATLAB to a C-style string and back again.
 *
 * Takes a string and returns a string in reverse order.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-1998 The MathWorks, Inc.
 *=========================================================*/
#include "mex.h"

void revord(char *input_buf, int buflen, char *output_buf)
{
  int   i;

  /* Reverse the order of the input string. */
  for(i=0;i<buflen-1;i++)
    *(output_buf+i) = *(input_buf+buflen-i-2);
}
```

In this example, the API function `mxCalloc` replaces `calloc`, the standard C function for dynamic memory allocation. `mxCalloc` allocates dynamic memory using MATLAB's 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`.

Below is the gateway function that calls the C computational routine `revord`.

```
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
    char *input_buf, *output_buf;
    int   buflen,status;

    /* Check for proper number of arguments. */
    if(nrhs!=1)
      mexErrMsgTxt("One input required.");
    else if(nlhs > 1)
      mexErrMsgTxt("Too many output arguments.");

    /* Input must be a string. */
    if ( mxIsChar(prhs[0]) != 1)
      mexErrMsgTxt("Input must be a string.");

    /* Input must be a row vector. */
    if (mxGetM(prhs[0])!=1)
      mexErrMsgTxt("Input must be a row vector.");

    /* Get the length of the input string. */
    buflen = (mxGetM(prhs[0]) * mxGetN(prhs[0])) + 1;

    /* Allocate memory for input and output strings. */
    input_buf=mxCalloc(buflen, sizeof(char));
    output_buf=mxCalloc(buflen, sizeof(char));
```

```
    /* Copy the string data from prhs[0] into a C string
     * input_ buf.
     * If the string array contains several rows, they are copied,
     * one column at a time, into one long string array.
     */
    status = mxGetString(prhs[0], input_buf, buflen);
    if(status != 0)
      mexWarnMsgTxt("Not enough space. String is truncated.");

    /* Call the C subroutine. */
    revord(input_buf, buflen, output_buf);

    /* Set C-style string output_buf to MATLAB mexFunction
       output*/
    plhs[0] = mxCreateString(output_buf);
    return;
}
```

The gateway function 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.

In this example, typing

```
x = 'hello world';
y = revord(x)
```

produces

```
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. For example, 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
```

This is the corresponding MEX-file C code.

```
#include "mex.h"

/*
 * xtimesy.c - example found in API guide
 *
 * Multiplies an input scalar times an input matrix and outputs a
 * matrix
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-1998 The MathWorks, Inc.
 */
```

```
/* $Revision: 1.5 $ */

void xtimesy(double x, double *y, double *z, int m, int n)
{
  int i,j,count=0;

  for (i=0; i<n; i++) {
    for (j=0; j<m; j++) {
      *(z+count) = x * *(y+count);
      count++;
    }
  }
}

/* The gateway function */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[])
{
  double *y,*z;
  double  x;
  int     status,mrows,ncols;

  /*  Check for proper number of arguments. */
  if(nrhs!=2)
    mexErrMsgTxt("Two inputs required.");
  if(nlhs!=1)
    mexErrMsgTxt("One output required.");

  /* Check to make sure the first input argument is a scalar. */
  if( !mxIsNumeric(prhs[0]) || !mxIsDouble(prhs[0]) ||
      mxIsEmpty(prhs[0])    || mxIsComplex(prhs[0]) ||
      mxGetN(prhs[0])*mxGetM(prhs[0])!=1 ) {
    mexErrMsgTxt("Input x must be a scalar.");
  }

  /*  Get the scalar input x. */
  x = mxGetScalar(prhs[0]);
```

```
    /*  Create a pointer to the input matrix y. */
    y = mxGetPr(prhs[1]);

    /*  Get the dimensions of the matrix input y. */
    mrows = mxGetM(prhs[1]);
    ncols = mxGetN(prhs[1]);

    /*  Set the output pointer to the output matrix. */
    plhs[0] = mxCreateDoubleMatrix(mrows,ncols, mxREAL);

    /*  Create a C pointer to a copy of the output matrix. */
    z = mxGetPr(plhs[0]);

    /*  Call the C subroutine. */
    xtimesy(x,y,z,mrows,ncols);

}
```

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`.

Manipulating Structures and Cell Arrays

Structures and cell arrays are new data types in MATLAB 5; for a discussion of the features of structures and cell arrays and the built-in functions MATLAB provides for manipulating them, refer to *Using MATLAB*. Like all other data types in MATLAB, structures and cell arrays can be passed into and out of C MEX-files.

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.

```
/* $Revision: 1.1 $ */
/*
=================================================================
 * phonebook.c
 * Example for illustrating how to manipulate structure and cell
 * arrays
 * Takes an (MxN) structure matrix and returns a new structure
 * (1-by-1) containing corresponding fields: for string input, it
 * will be (MxN) cell array; and for numeric (noncomplex, scalar)
 * input, it will be (MxN) vector of numbers with the same classID
 * as input, such as int, double, etc..
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-1998 The MathWorks, Inc.
 *=================================================================*/
#include "mex.h"
#include "string.h"
#define MAXCHARS 80   /* max length of string contained in each
                         field */

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

```
const char **fnames;       /* Pointers to field names */
const int  *dims;
mxArray    *tmp, *fout;
char       *pdata;
int        ifield, jstruct, *classIDflags;
int        NStructElems, nfields, ndim;

/* Check proper input and output. */
if(nrhs!=1)
  mexErrMsgTxt("One input required.");
else if(nlhs > 1)
  mexErrMsgTxt("Too many output arguments.");
else if(!mxIsStruct(prhs[0]))
  mexErrMsgTxt("Input must be a structure.");
/* Get input arguments. */
nfields = mxGetNumberOfFields(prhs[0]);
NStructElems = mxGetNumberOfElements(prhs[0]);
/* Allocate memory  for storing classIDflags. */
classIDflags = mxCalloc(nfields, sizeof(int));

/* Check empty field, proper data type, and data type
 consistency; and get classID for each field. */
 for(ifield=0; ifield<nfields; ifield++) {
 for(jstruct = 0; jstruct < NStructElems; jstruct++) {
   tmp = mxGetFieldByNumber(prhs[0], jstruct, ifield);
   if(tmp == NULL) {
   mexPrintf("%s%d\t%s%d\n", "FIELD: ", ifield+1, "STRUCT
             INDEX :", jstruct+1);
   mexErrMsgTxt("Above field is empty!");
   }
   if(jstruct==0) {
   if( !mxIsChar(tmp) && !mxIsNumeric(tmp)) {
       mexPrintf("%s%d\t%s%d\n", "FIELD: ", ifield+1, "STRUCT
                 INDEX :", jstruct+1);
       mexErrMsgTxt("Above field must have either string or
                 numeric data.");
   }
   classIDflags[ifield]=mxGetClassID(tmp);
   } else {
```

```
   if (mxGetClassID(tmp) != classIDflags[ifield]) {
       mexPrintf("%s%d\t%s%d\n", "FIELD: ", ifield+1,
                 "STRUCT INDEX :", jstruct+1);
      mexErrMsgTxt("Inconsistent data type in above field!");
   } else if(!mxIsChar(tmp) &&
      ((mxIsComplex(tmp) || mxGetNumberOfElements(tmp)!=1))){
     mexPrintf("%s%d\t%s%d\n", "FIELD: ", ifield+1,
               "STRUCT INDEX :", jstruct+1);
     mexErrMsgTxt("Numeric data in above field must be scalar
                   and noncomplex!");
   }
   }
}
}

/* Allocate memory  for storing pointers. */
fnames = mxCalloc(nfields, sizeof(*fnames));
/* Get field name pointers. */
for (ifield=0; ifield< nfields; ifield++){
fnames[ifield] = mxGetFieldNameByNumber(prhs[0],ifield);
}
/* Create a 1x1 struct matrix for output.  */
plhs[0] = mxCreateStructMatrix(1, 1, nfields, fnames);
mxFree(fnames);
ndim = mxGetNumberOfDimensions(prhs[0]);
dims = mxGetDimensions(prhs[0]);
for(ifield=0; ifield<nfields; ifield++) {
/* Create cell/numeric array. */
if(classIDflags[ifield] == mxCHAR_CLASS) {
    fout = mxCreateCellArray(ndim, dims);
}else {
    fout = mxCreateNumericArray(ndim, dims,
           classIDflags[ifield], mxREAL);
    pdata = mxGetData(fout);
}
/* Copy data from input structure array. */
for (jstruct=0; jstruct<NStructElems; jstruct++) {
    tmp = mxGetFieldByNumber(prhs[0],jstruct,ifield);
    if( mxIsChar(tmp)) {
    mxSetCell(fout, jstruct, mxDuplicateArray(tmp));
```

```
            }else {
            size_t     sizebuf;
            sizebuf = mxGetElementSize(tmp);
            memcpy(pdata, mxGetData(tmp), sizebuf);
            pdata += sizebuf;
            }
        }
        /* Set each field in output structure. */
        mxSetFieldByNumber(plhs[0], 0, ifield, fout);
        }
        mxFree(classIDflags);
        return;
```

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]
```

Handling Complex Data

Complex data from MATLAB is separated into real and imaginary parts. MATLAB's API provides two functions, `mxGetPr` and `mxGetPi`, that return pointers (of type `double *`) to the real and imaginary parts of your data.

This example takes two complex row vectors and convolves them.

```
/* $Revision: 1.5 $ */
/*=========================================================
 * convec.c
 * Example for illustrating how to pass complex data
 * from MATLAB to C and back again
 *
 * Convolves two complex input vectors.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-1998 The MathWorks, Inc.
 *=========================================================*/
#include "mex.h"

/* Computational subroutine */
void convec( double *xr, double *xi, int nx,
             double *yr, double *yi, int ny,
             double *zr, double *zi)
{
    int i,j;

    zr[0]=0.0;
    zi[0]=0.0;
    /* Perform the convolution of the complex vectors. */
    for(i=0; i<nx; i++) {
     for(j=0; j<ny; j++) {
       *(zr+i+j) = *(zr+i+j) + *(xr+i) * *(yr+j) - *(xi+i)
       * *(yi+j);
       *(zi+i+j) = *(zi+i+j) + *(xr+i) * *(yi+j) + *(xi+i)
       * *(yr+j);
     }
   }
}
```

Below is the gateway function that calls this complex convolution.

```
/* The gateway routine. */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
    double  *xr, *xi, *yr, *yi, *zr, *zi;
    int     rows, cols, nx, ny;

    /* Check for the proper number of arguments. */
    if(nrhs != 2)
      mexErrMsgTxt("Two inputs required.");
    if(nlhs > 1)
      mexErrMsgTxt("Too many output arguments.");
    /* Check that both inputs are row vectors. */
    if( mxGetM(prhs[0]) != 1 || mxGetM(prhs[1]) != 1 )
      mexErrMsgTxt("Both inputs must be row vectors.");
    rows = 1;
    /* Check that both inputs are complex. */
    if( !mxIsComplex(prhs[0]) || !mxIsComplex(prhs[1]) )
      mexErrMsgTxt("Inputs must be complex.\n");

    /* Get the length of each input vector. */
    nx = mxGetN(prhs[0]);
    ny = mxGetN(prhs[1]);


    /* Get pointers to real and imaginary parts of the inputs. */
    xr = mxGetPr(prhs[0]);
    xi = mxGetPi(prhs[0]);
    yr = mxGetPr(prhs[1]);
    yi = mxGetPi(prhs[1]);

    /* Create a new array and set the output pointer to it. */
    cols = nx + ny – 1;
    plhs[0] = mxCreateDoubleMatrix(rows, cols, mxCOMPLEX);
    zr = mxGetPr(plhs[0]);
    zi = mxGetPi(plhs[0]);
```

```
    /* Call the C subroutine. */
    convec(xr, xi, nx, yr, yi, ny, zr, zi);

    return;
}
```

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 5 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 5 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. Although from within MATLAB it is not currently possible to perform arithmetic or to call MATLAB functions that perform data manipulation on data of 8-, 16-, or 32-bit precision, you can display the data at the MATLAB prompt and save it in a MAT-file.

This example constructs a 2-by-2 matrix with unsigned 16-bit integers, doubles each element, and returns both matrices to MATLAB.

```
#include <string.h> /* Needed for memcpy() */
#include "mex.h"

/*
 * doubleelement.c - Example found in API Guide
 *
 * Constructs a 2-by-2 matrix with unsigned 16-bit integers,
 * doubles each element, and returns the matrix.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-1998 The MathWorks, Inc.
 */

/* $Revision: 1.4 $ */

#define NDIMS 2
#define TOTAL_ELEMENTS 4

/* The computational subroutine */
void dbl_elem(unsigned short *x)
{
  unsigned short scalar=2;
  int i,j;

  for(i=0;i<2;i++) {
    for(j=0;j<2;j++) {
      *(x+i+j) = scalar * *(x+i+j);
    }
  }
}
```

```
/* The gataway function */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
  const int dims[]={2,2};
  unsigned char *start_of_pr;
  unsigned short data[]={1,2,3,4};
  int bytes_to_copy;

  /* Call the computational subroutine. */
  dbl_elem(data);

  /* Create a 2-by-2 array of unsigned 16-bit integers. */
  plhs[0] = mxCreateNumericArray(NDIMS,dims,
            mxUINT16_CLASS,mxREAL);

  /* Populate the real part of the created array. */
  start_of_pr = (unsigned char *)mxGetPr(plhs[0]);
  bytes_to_copy = TOTAL_ELEMENTS * mxGetElementSize(plhs[0]);
  memcpy(start_of_pr,data,bytes_to_copy);
}
```

At the MATLAB prompt, entering

```
doubleelement
```

produces

```
ans =
     2     6
     8     4
```

The output of this function is a 2-by-2 matrix populated with unsigned 16-bit integers. You can view the contents of this matrix in MATLAB, but you cannot manipulate the data in any fashion.

Manipulating Multidimensional Numerical Arrays

Multidimensional numerical arrays are a new data type in MATLAB 5. For a discussion of the features of multidimensional numerical arrays and the built-in functions MATLAB provides to manipulate them, refer to *Using MATLAB*. Like all other data types in MATLAB, arrays can be passed into and

out of MEX-files written in C. 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.

This example takes an N-dimensional array of doubles and returns the indices for the nonzero elements in the array.

```
/*==========================================================
 * findnz.c
 * Example for illustrating how to handle N-dimensional arrays in
 * a MEX-file. NOTE: MATLAB uses 1-based indexing, C uses 0-based
 * indexing.
 *
 * Takes an N-dimensional array of doubles and returns the indices
 * for the non-zero elements in the array. findnz works
 * differently than the FIND command in MATLAB in that it returns
 * all the indices in one output variable, where the column
 * element contains the index for that dimension.
 *
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-98 by The MathWorks, Inc.
 * All Rights Reserved.
 *========================================================*/

/* $Revision: 1.2 $ */
#include "mex.h"

/* If you are using a compiler that equates NaN to zero, you must
 * compile this example using the flag –DNAN_EQUALS_ZERO. For
 * example:
 *
 *     mex –DNAN_EQUALS_ZERO findnz.c
 *
 * This will correctly define the IsNonZero macro for your
   compiler. */
```

```
#if NAN_EQUALS_ZERO
#define IsNonZero(d) ((d)!=0.0 || mxIsNaN(d))
#else
#define IsNonZero(d) ((d)!=0.0)
#endif

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
    /* Declare variables. */
    int elements, j, number_of_dims, cmplx;
    int nnz=0, count=0;
    double *pr, *pi, *pind;
    const int  *dim_array;

   /* Check for proper number of input and output arguments. */
    if (nrhs != 1) {
    mexErrMsgTxt("One input argument required.");
    }
    if (nlhs > 1){
    mexErrMsgTxt("Too many output arguments.");
    }

    /* Check data type of input argument. */
    if (!(mxIsDouble(prhs[0]))) {
      mexErrMsgTxt("Input array must be of type double.");
    }

    /* Get the number of elements in the input argument. */
    elements = mxGetNumberOfElements(prhs[0]);
    /* Get the data. */
    pr = (double *)mxGetPr(prhs[0]);
    pi = (double *)mxGetPi(prhs[0]);
    cmplx = ((pi==NULL) ? 0 : 1);
```

```
    /* Count the number of non-zero elements to be able to allocate
        the correct size for output variable. */
     for(j=0;j<elements;j++){
     if(IsNonZero(pr[j]) || (cmplx && IsNonZero(pi[j]))) {
        nnz++;
     }
     }
     /* Get the number of dimensions in the input argument.
        Allocate the space for the return argument */
number_of_dims = mxGetNumberOfDimensions(prhs[0]);
plhs[0] = mxCreateDoubleMatrix(nnz, number_of_dims, mxREAL);
pind = mxGetPr(plhs[0]);

/* Get the number of dimensions in the input argument. */
dim_array = mxGetDimensions(prhs[0]);

/* Fill in the indices to return to MATLAB. This loops through the
* elements and checks for non-zero values. If it finds a non-zero
* value, it then calculates the corresponding MATLAB indices and
* assigns them into the output array. The 1 is added to the
* calculated index because MATLAB is 1-based and C is 0-based. */
for(j=0;j<elements;j++) {
if(IsNonZero(pr[j]) || (cmplx && IsNonZero(pi[j]))) {
   int temp=j;
   int k;
   for (k=0;k<number_of_dims;k++){
   pind[nnz*k+count]=((temp % (dim_array[k])) +1);
   temp/=dim_array[k];
   }
   count++;
  }
  }
}
```

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 5 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, refer to "The MATLAB Array" section in Chapter 1 of this guide.

This example illustrates how to populate a sparse matrix.

```
/*==========================================================
* fulltosparse.c
* This example demonstrates how to populate a sparse
* matrix.  For the purpose of this example, you must pass in a
* nonsparse, 2-dimensional argument of type double.

* Comment: You might want to modify this MEX-file so that you can
* use it to read large sparse data sets into MATLAB.
*
* This is a MEX-file for MATLAB.
* Copyright (c) 1984-98 by The MathWorks, Inc.
* All rights reserved.
*========================================================*/

/* $Revision: 1.2 $ */

#include <math.h> /* Needed for the ceil() prototype */
#include "mex.h"

/* If you are using a compiler that equates NaN to be zero, you
 * must compile this example using the flag –DNAN_EQUALS_ZERO.
 * For example:
 *
 *     mex –DNAN_EQUALS_ZERO fulltosparse.c
 *
 * This will correctly define the IsNonZero macro for your C
 * compiler. */

#if NAN_EQUALS_ZERO
#define IsNonZero(d) ((d)!=0.0 || mxIsNaN(d))
#else
#define IsNonZero(d) ((d)!=0.0)
#endif

#define PERCENT_SPARSE .20

void mexFunction(
        int nlhs,       mxArray *plhs[],
```

```
        int nrhs, const mxArray *prhs[]
        )
{
    /* Declare variable. */
    int j, k, m, n, nzmax, *irs, *jcs, cmplx, isfull;
    double *pr, *pi, *si, *sr;

   /* Check for proper number of input and output arguments. */
    if (nrhs != 1) {
    mexErrMsgTxt("One input argument required.");
    }
    if(nlhs > 1){
    mexErrMsgTxt("Too many output arguments.");
    }

    /* Check data type of input argument.  */
    if (!(mxIsDouble(prhs[0]))){
    mexErrMsgTxt("Input argument must be of type double.");
    }

    if (mxGetNumberOfDimensions(prhs[0]) != 2){
    mexErrMsgTxt("Input argument must be two dimensional\n");
    }

    /* Get the size and pointers to input data. */
    m  = mxGetM(prhs[0]);
    n  = mxGetN(prhs[0]);
    pr = mxGetPr(prhs[0]);
    pi = mxGetPi(prhs[0]);
    cmplx = (pi==NULL ? 0 : 1);

    /* Allocate space for sparse matrix. */
    /* NOTE:  Assume at most 20% of the data is sparse.  Use ceil
     * to cause it to round up. */
    nzmax = (int)ceil((double)m*(double)n*PERCENT_SPARSE);
   /* NOTE: The maximum number of non-zero elements cannot be less
       than the number of columns in the matrix. */
    if (n>nzmax){
    nzmax = n;
    }
```

```
    plhs[0] = mxCreateSparse(m, n, nzmax, cmplx);
    sr  = mxGetPr(plhs[0]);
    si  = mxGetPi(plhs[0]);
    irs = mxGetIr(plhs[0]);
    jcs = mxGetJc(plhs[0]);

    /* Copy non-zeros. */
    k = 0;
    isfull = 0;
    for (j=0; (j<n ); j++) {
    int i;
    jcs[j] = k;
    for (i=0; (i<m ); i++) {
        if (IsNonZero(pr[i]) || (cmplx && IsNonZero(pi[i]))) {
        /* Check to see if non-zero element will fit in
        * allocated output array.  If not, set flag and
        * print warning. */
        if (k>=nzmax){
            isfull=1;
            mexWarnMsgTxt("Truncating output, more than 20%% of
                           input is non-zero data.");
            break;
        }
        sr[k] = pr[i];
        if (cmplx){
        si[k] = pi[i];
        }
        irs[k] = i;
        k++;
        }
    }
    if (isfull)
    break;
    pr += m;
    pi += m;
    }
    jcs[n] = k;
}
```

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 MATLAB Functions and Other User-Defined Functions from Within a MEX-File

It is possible to call MATLAB functions, operators, M-files, and other MEX-files from within your C source code by using the API function `mexCallMATLAB`. This 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.

```
/* $Revision: 1.1 $ */
/*=================================================================
 * sincall.c
 *
 * Example for illustrating how to use mexCallMATLAB
 *
 * Creates an mxArray and passes its associated pointers (in
 * this demo, only pointer to its real part, pointer to number of
 * rows, pointer to number of columns) to subfunction fill() to
 * get data filled up, then calls mexCallMATLAB to calculate sin
 * function and plot the result.
 *
```

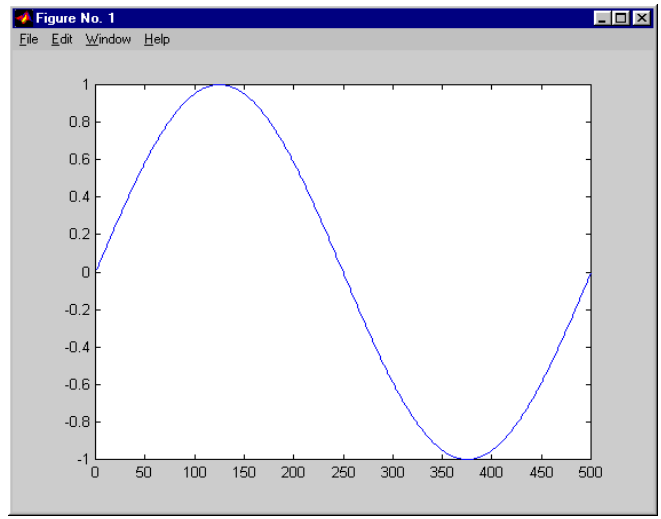
```
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-1998 The MathWorks, Inc.
 *=================================================================*/
#include "mex.h"
#define MAX 1000

/* Subroutine for filling up data */
void fill( double *pr, int *pm, int *pn, int max )
{
    int i;
    /* You can fill up to max elements, so (*pr)<=max. */
    *pm = max/2;
    *pn = 1;
    for (i=0; i < (*pm); i++)
      pr[i]=i*(4*3.14159/max);
}

/* Gateway function */
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
    int     m, n, max=MAX;
    mxArray *rhs[1], *lhs[1];

    rhs[0] = mxCreateDoubleMatrix(max, 1, mxREAL);

    /* Pass the pointers and let fill() fill up data. */
    fill(mxGetPr(rhs[0]), &m, &n, MAX);
    mxSetM(rhs[0], m);
    mxSetN(rhs[0], n);


    /* Get the sin wave and plot it. */
    mexCallMATLAB(1, lhs, 1, rhs, "sin");
    mexCallMATLAB(0, NULL, 1, lhs, "plot");
```

```
    /* Clean up allocated memory. */
    mxDestroyArray(rhs[0]);
    mxDestroyArray(lhs[0]);

    return;
}
```

Running this example

```
sincall
```

displays the results



Note: It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. For example, if you create an M-file that returns two variables but only assigns one of them a value,

```
function [a,b]=foo[c]
a=2*c;
```

you'll get this warning message in MATLAB:

```
Warning: One or more output arguments not assigned during call
to 'foo'.
```

If you then call `foo` using `mexCallMATLAB`, the unassigned output variable will now be of type `mxUNKNOWN_CLASS`.

Advanced Topics

These sections cover advanced features of MEX-files that you can use when your applications require sophisticated MEX-files.

Help Files

Because the MATLAB interpreter chooses the 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 MEX-files. The MATLAB `help` command will automatically find and display the appropriate M-file when help is requested and the interpreter will find and execute the corresponding MEX-file when the function is invoked.

Linking Multiple Files

It is possible to combine several object files and to use object file libraries when building MEX-files. To do so, simply list the additional files with their full extension, separated by spaces. For example, on the PC

```
mex circle.c square.obj rectangle.c shapes.lib
```

is a legal command that operates on the `.c`, `.obj`, and `.lib` files to create a MEX-file called `circle.dll`, where `dll` is the extension corresponding to the MEX-file type on the PC. The name of the resulting MEX-file is taken from the first file in the list.

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 `mex` to combine your object files into a MEX-file. This way you can ensure that your source files are recompiled only when necessary.

Variable Scope

Unlike M-file functions, MEX-file functions 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 `mexGetArray` and `mexPutArray` routines to get and put variables into the caller's workspace.

Memory Management

Memory management within MEX-files is not unlike memory management for regular C or Fortran applications. However, there are special considerations because the MEX-file must exist within the context of a larger application, i.e., MATLAB itself.

Automatic Cleanup of Temporary Arrays

When a MEX-file returns to MATLAB, it gives to MATLAB the results of its computations in the form of the left-hand side arguments – the `mxArrays` contained within the `plhs[]` list. Any `mxArrays` created by the MEX-file that are not in this list are automatically destroyed. In addition, any memory allocated with `mxCalloc`, `mxMalloc`, or `mxRealloc` during the MEX-file's execution is automatically freed.

In general, we recommend that MEX-files destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient for the MEX-file to perform this cleanup than to rely on the automatic mechanism. However, there are several circumstances in which the MEX-file will not reach its normal return statement. The normal return will not be reached if:

- A call to `mexErrMsgTxt` occurs.
- A call to `mexCallMATLAB` occurs and the function being called creates an error. (A MEX-file can trap such errors by using `mexSetTrapFlag`, but not all MEX-files would necessarily need to trap errors.)
- The user interrupts the MEX-file's execution using Ctrl-C.
- The MEX-file runs out of memory. When this happens, MATLAB's out-of-memory handler will immediately terminate 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 MATLAB's automatic cleanup by calling `mexMakeArrayPersistent` or `mexMakeMemoryPersistent`. However, if a 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. In order to prevent this from happening, a MEX-file that

creates persistent objects should register a function, using `mexAtExit`, which will dispose 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 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` will destroy 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." 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 may cause your MEX-file to crash.

Although persistent hybrid arrays are viable, we recommend avoiding their use wherever possible.

How to Debug C Language MEX-Files

On most platforms, it is now possible to debug MEX-files while they are running within MATLAB. Complete source code debugging, including setting breakpoints, examining variables, and stepping through the source code line-by-line, is now available.

Note: The section, "Troubleshooting," in Chapter 8 provides additional information on isolating problems with MEX-files.

To debug a MEX-file from within MATLAB, you must first compile the MEX-file with the –g option to `mex`.

```
mex –g filename.c
```

Debugging on UNIX

You will need to start MATLAB from within a debugger. To do this, specify the name of the debugger you want to use with the –D option when starting MATLAB. For example, to use `dbx`, the UNIX debugger, type

```
matlab –Ddbx
```

Once the debugger loads MATLAB into memory, you can start it by issuing a "run" command. Now, from within MATLAB, enable MEX-file debugging by typing

```
dbmex on
```

at the MATLAB prompt. Then, run the MEX-file that you want to debug as you would ordinarily do (either directly or by means of some other function or script). Before executing the MEX-file, you will be returned to the debugger.

You may need to tell the debugger where the MEX-file was loaded or the name of the MEX-file, in which case MATLAB will display the appropriate command for you to use. At this point, you are ready to start debugging. You can list the source code for your MEX-file and set breakpoints in it. It is often convenient to set one at `mexFunction` so that you stop at the beginning of the gateway function. To proceed from the breakpoint, issue a "continue" command to the debugger.

Once you hit one of your breakpoints, you can make full use of any facilities that your debugger provides to examine variables, display memory, or inspect registers. Refer to the documentation provided with your debugger for information on its use.

If you are at the MATLAB prompt and want to return control to the debugger, you can issue the command

```
dbmex stop
```

which allows you to gain access to the debugger so that you can set additional breakpoints or examine source code. To resume execution, issue a "continue" command to the debugger.

Debugging on Windows

The following sections provide instructions on how to debug on Microsoft Windows systems using various compilers.

Microsoft Compiler. If you are using the Microsoft compiler:

1. Start the Microsoft Development Studio (Version 4.2) or the Microsoft Visual Studio (Version 5) by typing at the DOS prompt

   ```
   msdev filename.dll
   ```

2. In the Microsoft environment, from the **Build** menu (Version 4.2) or the **Project** menu (Version 5.0), select **Settings**. In the window that opens, select the **Debug** tab. This options window contains edit boxes. In the edit box labeled **Executable for debug session**, enter the full path to where MATLAB 5 resides. All other edit boxes should be empty.

3. Open the source files and set a break point on the desired line of code by right-clicking with your mouse on the line of code.

4. From the **Build** menu, select **Debug**, and click **Go**.

5. You will now be able to run your MEX-file in MATLAB and use the Microsoft debugging environment. For more information on how to debug in the Microsoft environment, see the Microsoft Development Studio or Microsoft Visual Studio documentation.

Watcom Compiler. If you are using the Watcom compiler:

1. Start the debugger by typing on the DOS command line

   ```
   WDW
   ```

2. The Watcom Debugger starts and a **New Program** window opens. In this window type the full path to MATLAB. For example,

   ```
   c:\matlab\bin\matlab.exe
   ```

   Then click **OK**.

3. From the **Break** menu, select **On Image Load** and type the name of your MEX-file DLL in capital letters. For example,

   ```
   YPRIME
   ```

   Then select **ADD** and click **OK** to close the window.

4. From the **Run** menu, select **GO**. This should start MATLAB.

5. When MATLAB starts, in the command window change directories to where your MEX-file resides and run your MEX-file. If a message similar to the following appears,

   ```
   LDR: Automatic DLL Relocation in matlab.exe
   LDR: DLL filename.dll base <number> relocated due to collision
   with matlab.exe
   ```

   ignore the message and click **OK**.

6. Open the file you want to debug and set breakpoints in the source code.

4

Creating Fortran MEX-Files

Fortran MEX-Files

Fortran MEX-files are built by using the `mex` script to compile your Fortran source code with additional calls to API routines.

Directory Organization

This table lists the location of the files on your disk that are associated with the creation of Fortran language MEX-files.

Platform	Directory
Windows	`<matlab>\extern`
UNIX	`<matlab>/extern`
	where: `<matlab>` is the MATLAB root directory

Appendix B, "Directory Organization," describes the API-related directories and files.

MEX-Files and Data Types

MEX-files in Fortran can only create double-precision data and strings (unlike their C counterparts, which can create any data type supported by MATLAB). You can treat Fortran MEX-files, once compiled, exactly like M-functions.

The Components of a Fortran MEX-File

This section discusses the specific elements needed in a Fortran MEX-file. The source code for a Fortran MEX-file, like the C MEX-file, consists of two distinct parts:

- A *computational routine* that contains the code for performing the computations that you want implemented in the MEX-file. Computations can be numerical computations as well as inputting and outputting data.
- A *gateway routine* that interfaces the computational routine with MATLAB by the entry point `mexFunction` and its parameters `prhs`, `nrhs`, `plhs`, `nlhs`, where `prhs` is an array of right-hand input arguments, `nrhs` is the number of right-hand input arguments, `plhs` is an array of left-hand output

arguments, and `nlhs` is the number of left-hand output arguments. The gateway calls the computational routine as a subroutine.

The computational and gateway routines may be separate or combined. Figure 4-1 shows how inputs enter an API function, what functions the gateway function performs, and how output returns to MATLAB.

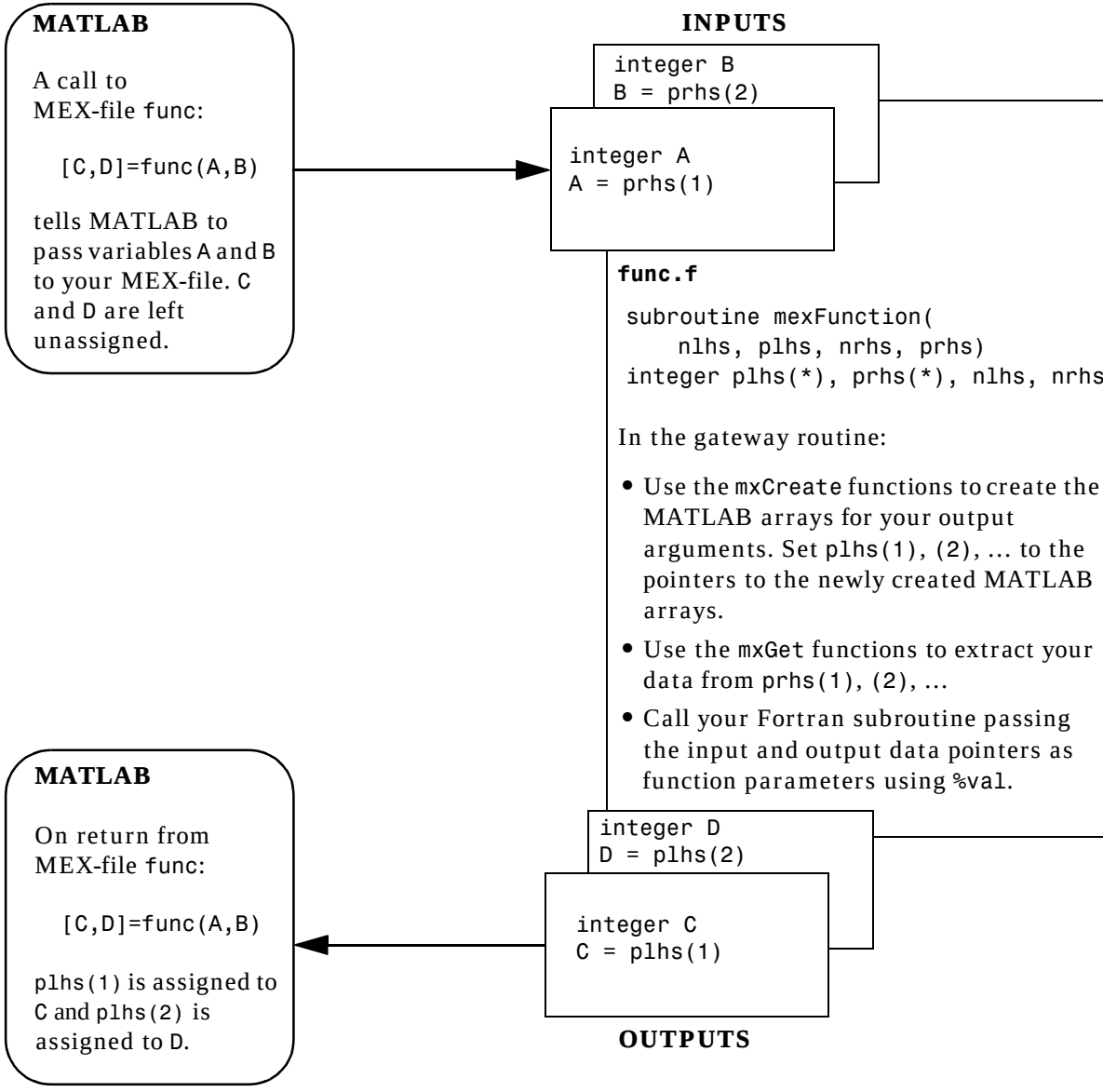


Figure 4-1: Fortran MEX Cycle

The Pointer Concept

The MATLAB API works with a unique data type, the `mxArray`. Because there is no way to create a new data type in Fortran, MATLAB passes a special identifier, called a pointer, to a Fortran program. You can get information about an `mxArray` by passing this pointer to various API functions called "Access Routines". These access routines allow you to get a native Fortran data type containing exactly the information you want, i.e., the size of the `mxArray`, whether or not it is a string, or its data contents.

There are several implications when using pointers in Fortran:

- The `%val` construct.

  If your Fortran compiler supports the `%val` construct, then there is one type of pointer you can use without requiring an access routine, namely a pointer to 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.

  If your Fortran compiler does not support the `%val` construct, you must use the `mxCopy__` routines (e.g., `mxCopyPtrToReal8`) to access the contents of the pointer. For more information about the `%val` construct and an example, see the section, "The %val Construct," in this chapter.

- Variable declarations.

  To use pointers properly, you must declare them to be the correct size. On DEC Alpha and 64-bit SGI machines, all pointers should be declared as `integer*8`. On all other platforms, pointers should be declared as `integer*4`.

  If your Fortran compiler supports preprocessing with the C preprocessor, you can use the preprocessing stage to map pointers to the appropriate declaration. In UNIX, see the examples ending with `.F` in the `examples` directory for a possible approach.

Caution: Declaring a pointer to be the incorrect size can cause your program to crash.

The Gateway Routine

The entry point to the gateway subroutine must be named `mexFunction` and must contain these parameters:

```
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer plhs(*), prhs(*)
integer nlhs, nrhs
```

Note: Fortran is case insensitive. This document uses mixed case function names for ease of reading.

In a Fortran MEX-file, the parameters `nlhs` and `nrhs` contain the number of left- and right-hand arguments with which the MEX-file is invoked. `prhs` is a length `nrhs` array that contains pointers to the right-hand side inputs to the MEX-file, and `plhs` is a length `nlhs` array that contains pointers to the left-hand side outputs that your Fortran function generates.

In the syntax of the MATLAB language, functions have the general form

```
[a,b,c,…] = fun(d,e,f,…)
```

where the ellipsis (…) denotes additional terms of the same format. The `a,b,c,…` are left-hand arguments and the `d,e,f,…` are right-hand arguments.

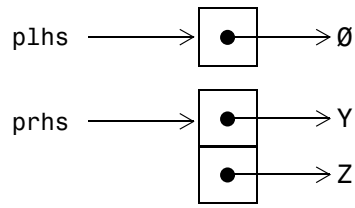
As an example of the gateway routine, consider invoking a MEX-file from the MATLAB workspace with the command

```
x = fun(y,z);
```

the MATLAB interpreter calls `mexFunction` with the arguments

`nlhs = 1`

`nrhs = 2`



plhs is a 1-element C array where the single element is a null pointer. prhs is a 2-element C array where the first element is a pointer to an mxArray named Y and the second element is a pointer to an mxArray named Z.

The parameter plhs points at nothing because the output x is not created until the subroutine executes. It is the responsibility of the gateway routine to create an output array and to set a pointer to that array in plhs(1). If plhs(1) is left unassigned, MATLAB prints a warning message stating that no output has been assigned.

Note: It is possible to return an output value even if nlhs = 0. This corresponds to returning the result in the ans variable.

The gateway routine should validate the input arguments and call mexErrMsgTxt if anything is amiss. This step includes checking the number, type, and size of the input arrays as well as examining the number of output arrays. The examples included later in this section illustrate this technique.

The mx functions provide a set of access methods (subroutines) for manipulating MATLAB arrays. These functions are fully documented in the online API reference pages. The mx prefix is shorthand for mxArray and it means that the function enables you to access and/or manipulate some of the information in the MATLAB array. For example, mxGetPr gets the real data from the MATLAB array. Additional routines are provided for transferring data between MATLAB arrays and Fortran arrays.

The gateway routine must call mxCreateFull, mxCreateSparse, or mxCreateString to create MATLAB arrays of the required sizes in which to return the results. The return values from these calls should be assigned to the appropriate elements of plhs.

The gateway routine may call mxCalloc to allocate temporary work arrays for the computational routine if it needs them.

The gateway routine should call the computational routine to perform the desired calculations or operations. There are a number of additional routines that MEX-files can use. These routines are distinguished by the initial characters mex, as in mexCallMATLAB and mexErrMsgTxt.

When a MEX-file completes its task, it returns control to MATLAB. Any MATLAB arrays that are created by the MEX-file that are not returned to MATLAB through the left-hand side arguments are automatically destroyed.

The %val Construct

The `%val` construct is supported by most, but not all, Fortran compilers. DIGITAL Visual Fortran *does* support the construct. `%val` causes the value of the variable, rather than the address of the variable, to be passed to the subroutine. If you are using a Fortran compiler that does not support the `%val` construct, you must copy the array values into a temporary true Fortran array using special routines. 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` will not work correctly.

Examples of Fortran MEX-Files

The next sections of this chapter include examples of different MEX-files. The MATLAB 5 API provides a set of routines for Fortran 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.

Note to UNIX Users: The example Fortran files in the directory `<matlab>/extern/examples/refbook` have extensions `.F` and `.f`. The distinction between these extensions is that the `.F` files need to be preprocessed.

Note: You can find the most recent versions of the example programs from this chapter at the anonymous FTP server,

`ftp.mathworks.com/pub/tech-support/library/matlab5/extern/examples/refbook`

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
```

Below is the same function written in the MEX-file format.

```
C-----------------------------------------------------------------
C     timestwo.f
C
C     Multiply the input argument by 2.
```

```
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-1998 The MathWorks, Inc.
C     $Revision: 1.6 $

      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC Alpha
C     and the SGI 64-bit platforms.
C
      integer plhs(*), prhs(*)
      integer mxGetPr, mxCreateFull
      integer x_pr, y_pr
C-----------------------------------------------------------------
C

      integer nlhs, nrhs
      integer mxGetM, mxGetN, mxIsNumeric
      integer m, n, size
      real*8  x, y

C     Check for proper number of arguments.
      if(nrhs .ne. 1) then
         call mexErrMsgTxt('One input required.')
      elseif(nlhs .ne. 1) then
         call mexErrMsgTxt('One output required.')
      endif

C     Get the size of the input array.
      m = mxGetM(prhs(1))
      n = mxGetN(prhs(1))
      size = m*n

C     Check to ensure the input is a number.
      if(mxIsNumeric(prhs(1)) .eq. 0) then
         call mexErrMsgTxt('Input must be a number.')
      endif
```

```
C     Create matrix for the return argument.
      plhs(1) = mxCreateFull(m, n, 0)
      x_pr = mxGetPr(prhs(1))
      y_pr = mxGetPr(plhs(1))
      call mxCopyPtrToReal8(x_pr, x, size)

C     Call the computational subroutine.
      call timestwo(y, x)

C     Load the data into y_pr, which is the output to MATLAB.
      call mxCopyReal8ToPtr(y, y_pr, size)

      return
      end

      subroutine timestwo(y, x)
      real*8 x, y
C
      y = 2.0 * x
      return
      end
```

To compile and link this example source file, at the MATLAB prompt type

```
mex timestwo.f
```

This carries out the necessary steps to create the 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.

```
x = 2;
y = timestwo(x)
y =
     4
```

Passing Strings

Passing strings from MATLAB to a Fortran MEX-file is straightforward. This program accepts a string and returns the characters in reverse order.

```
C     $Revision: 1.9 $
C=================================================================
C
C     revord.f
C     Example for illustrating how to copy string data from
C     MATLAB to a Fortran-style string and back again.
C
C     Takes a string and returns a string in reverse order.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-1998 The MathWorks, Inc.
C=================================================================

      subroutine revord(input_buf, strlen, output_buf)
      character  input_buf(*), output_buf(*)
      integer    i, strlen
      do 10 i=1,strlen
         output_buf(i) = input_buf(strlen-i+1)
 10   continue
      return
      end
```

Below is the gateway function that calls the computational routine.

```
C     The gateway routine
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
      integer nlhs, nrhs
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC Alpha
C     and the SGI 64-bit platforms.
C
      integer plhs(*), prhs(*)
      integer mxCreateString, mxGetString
C-----------------------------------------------------------------
C
```

```
      integer  mxGetM, mxGetN, mxIsString
      integer  status, strlen
      character*100 input_buf, output_buf

C     Check for proper number of arguments.
      if (nrhs .ne. 1) then
         call mexErrMsgTxt('One input required.')
      elseif (nlhs .gt. 1) then
         call mexErrMsgTxt('Too many output arguments.')
C     The input  must be a string.
      elseif(mxIsString(prhs(1)) .ne. 1) then
         call mexErrMsgTxt('Input must be a string.')
C     The input must be a row vector.
      elseif (mxGetM(prhs(1)) .ne. 1) then
         call mexErrMsgTxt('Input must be a row vector.')
      endif

C     Get the length of the input string.
      strlen = mxGetM(prhs(1))*mxGetN(prhs(1))

C     Get the string contents (dereference the input integer).
      status = mxGetString(prhs(1), input_buf, 100)

C     Check if mxGetString is successful.
      if (status .ne. 0) then
        call mexErrMsgTxt('String length must be less than 100.')
      endif

C     Initialize outbuf_buf to blanks. This is necessary on some
C     compilers.

      output_buf = ' '

C     Call the computational subroutine.
      call revord(input_buf, strlen, output_buf)
```

```
C     Set output_buf to MATLAB mexFunction output.
      plhs(1) = mxCreateString(output_buf)

      return
      end
```

After checking for the correct number of inputs, this MEX-file gateway function verifies that the input was either a row or column 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 by using `mxCopyPtrToCharacter`. In fact, `mxCopyPtrToCharacter` works only with MAT-files. (For more information about MAT-files, see Chapter 5, "Data Export and Import.")

For an input string

```
x = 'hello world';
```

typing

```
y = revord(x)
```

produces

```
y =
dlrow olleh
```

Passing Arrays of Strings

Passing arrays of strings involves a slight complication from the previous example in the "Passing Strings" section of this chapter. Because MATLAB stores elements of a matrix by column instead of by row, it is essential that the size of the string array 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. Inside MATLAB, call this function by typing

```
passstr;
```

You will get the matrix `mystring` of size 5-by-15. There are some manipulations that need to be done here. The original string matrix is of the size 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` from the MEX-file. After the matrix is put into MATLAB, the matrix must be transposed.

```
C     $Revision: 1.6 $
C=================================================================
C
C     passstr.f
C     Example for illustrating how to pass a character matrix
C     from Fortran to MATLAB.
C
C     Passes a string array/character matrix into MATLAB as
C     output arguments rather than placing it directly into the
C     workspace.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-1998 The MathWorks, Inc.

C=================================================================
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC Alpha
C     and the SGI 64-bit platforms.
C
      integer plhs(*), prhs(*)
      integer p_str, mxCreateString
C-----------------------------------------------------------------
C
      integer nlhs, nrhs
      integer i
      character*75 thestring
      character*15 string(5)
```

```
C     Create the string to passed into MATLAB.
      string(1) = 'MATLAB         '
      string(2) = 'The Scientific '
      string(3) = 'Computing      '
      string(4) = 'Environment    '
      string(5) = '   by TMW, Inc.'

C     Concatenate the set of 5 strings into a long string.
      thestring = string(1)
      do 10 i = 2, 6
         thestring = thestring(:((i-1)*15)) // string(i)
 10   continue

C     Create the string matrix to be passed into MATLAB.
C     Set the matrix size to be M=15 and N=5.
      p_str = mxcreatestring(thestring)
      call mxSetM(p_str, 15)
      call mxSetN(p_str, 5)

C     Transpose the resulting matrix in MATLAB.
      call mexCallMATLAB(1, plhs, 1, p_str, 'transpose')

      return
      end
```

Typing

```
passstr
```

at the MATLAB prompt produces this result

```
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, and you can create new MATLAB arrays from within your MEX-file by using `mxCreateFull`.

This example takes a real 2-by-3 matrix and squares each element.

```
C-------------------------------------------------------------
C
C     matsq.f
C
C     Squares the input matrix

C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-1998 The MathWorks, Inc.
C     $Revision: 1.7 $
C-------------------------------------------------------------

      subroutine matsq(y, x, m, n)
      real*8 x(m,n), y(m,n)
      integer m, n
C
      do 20 i=1,m
         do 10 j=1,n
            y(i,j)= x(i,j)**2
 10      continue
 20   continue
      return
      end
```

This is the gateway routine that calls the computational subroutine.

```
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC Alpha
C     and the SGI 64-bit platforms.
C
      integer plhs(*), prhs(*)
      integer mxCreateFull, mxGetPr
      integer x_pr, y_pr
C-----------------------------------------------------------------
C
      integer nlhs, nrhs
      integer mxGetM, mxGetN, mxIsNumeric
      integer m, n, size
      real*8  x(1000), y(1000)

C     Check for proper number of arguments.
      if(nrhs .ne. 1) then
         call mexErrMsgTxt('One input required.')
      elseif(nlhs .ne. 1) then
         call mexErrMsgTxt('One output required.')
      endif

C     Get the size of the input array.
      m = mxGetM(prhs(1))
      n = mxGetN(prhs(1))
      size = m*n

C     Column * row should be smaller than 1000.
      if(size.gt.1000) then
         call mexErrMsgTxt('Row * column must be <= 1000.')
      endif

C     Check to ensure the array is numeric (not strings).
      if(mxIsNumeric(prhs(1)) .eq. 0) then
         call mexErrMsgTxt('Input must be a numeric array.')
      endif
```

```
C     Create matrix for the return argument.
      plhs(1) = mxCreateFull(m, n, 0)
      x_pr = mxGetPr(prhs(1))
      y_pr = mxGetPr(plhs(1))
      call mxCopyPtrToReal8(x_pr, x, size)

C     Call the computational subroutine.
      call matsq(y, x, m, n)

C     Load the data into y_pr, which is the output to MATLAB.
      call mxCopyReal8ToPtr(y, y_pr, size)

      return
      end
```

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 for the argument returned from the computational subroutine. The input matrix data is then copied to a Fortran matrix by using `mxCopyPtrToReal8`. Now the computational subroutine can be called, and the return argument can then be placed into `y_pr`, the pointer to the output, using `mxCopyReal8ToPtr`.

For a 2-by-3 real matrix,

```
x = [1 2 3; 4 5 6];
```

typing

```
y = matsq(x)
```

produces this result

```
y =
     1     4     9
    16    25    36
```

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 right-hand side (input) variable. Accordingly, `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.

For example, here's a routine that multiplies an input scalar times an input scalar or matrix. This is the Fortran code for the computational subroutine.

```
      subroutine xtimesy(x, y, z, m, n)
      real*8  x, y(3,3), z(3,3)
      integer m, n
      do 20 i=1,m
         do 10 j=1,n
            z(i,j)=x*y(i,j)
10       continue
20    continue
      return
      end
```

Below is the gateway routine that calls `xtimesy`, the computation subroutine that multiplies a scalar by a scalar or matrix.

```
C-----------------------------------------------------------------
C
C     xtimesy.f
C
C     Multiply the first input by the second input.

C     This is a MEX file for MATLAB.
C     Copyright (c) 1984-1998 The MathWorks, Inc.
C     $Revision: 1.6 $

      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC Alpha
C     and the SGI 64-bit platforms.
C
      integer plhs(*), prhs(*)
      integer mxCreateFull
      integer x_pr, y_pr, z_pr
C-----------------------------------------------------------------
C
```

```
      integer nlhs, nrhs
      integer m, n, size
      integer mxGetM, mxGetN, mxIsNumeric
      real*8  x, y(3,3), z(3,3)

C     Check for proper number of arguments.
      if (nrhs .ne. 2) then
         call mexErrMsgTxt('Two inputs required.')
      elseif (nlhs .ne. 1) then
         call mexErrMsgTxt('One output required.')
      endif

C     Check to see both inputs are numeric.
      if (mxIsNumeric(prhs(1)) .ne. 1) then
         call mexErrMsgTxt('Input # 1 is not a numeric.')
      elseif (mxIsNumeric(prhs(2)) .ne. 1) then
         call mexErrMsgTxt('Input #2 is not a numeric array.')
      endif

C     Check that input #1 is a scalar.
      m = mxGetM(prhs(1))
      n = mxGetN(prhs(1))
      if(n .ne. 1 .or. m .ne. 1) then
         call mexErrMsgTxt('Input #1 is not a scalar.')
      endif

C     Get the size of the input matrix.
      m = mxGetM(prhs(2))
      n = mxGetN(prhs(2))
      size = m*n

C     Create matrix for the return argument.
      plhs(1) = mxCreateFull(m, n, 0)
      x_pr = mxGetPr(prhs(1))
      y_pr = mxGetPr(prhs(2))
      z_pr = mxGetPr(plhs(1))
```

```
C     Load the data into Fortran arrays.
      call mxCopyPtrToReal8(x_pr, x, 1)
      call mxCopyPtrToReal8(y_pr, y, size)

C     Call the computational subroutine.
      call xtimesy(x, y, z, m, n)

C     Load the output into a MATLAB array.
      call mxCopyReal8ToPtr(z, z_pr, size)

      return
      end
```

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)`.

For an input scalar `x` and a real 3-by-3 matrix,

```
x = 3; y = ones(3);
```

typing

```
z = xtimesy(x, y)
```

yields this result

```
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 contains the real data and one contains the imaginary data. The API provides two functions, `mxCopyPtrToComplex16` and `mxCopyComplex16ToPtr`, which allow you to copy the MATLAB data to a native `complex*16` Fortran array.

This example takes two complex vectors (of length 3) and convolves them.

```
C     $Revision: 1.9 $
C=================================================================
C
C     convec.f
C     Example for illustrating how to pass complex data from
C     MATLAB to FORTRAN (using COMPLEX data type) and back
C     again.
C
C     Convolves two complex input vectors.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-1998 The MathWorks, Inc.
C=================================================================
C
C     Computational subroutine
      subroutine convec(x, y, z, nx, ny)
      complex*16 x(*), y(*), z(*)
      integer nx, ny

C     Initialize the output array.
      do 10 i=1,nx+ny-1
         z(i) = (0.0,0.0)
 10   continue

      do 30 i=1,nx
         do 20 j=1,ny
            z(i+j-1) = z(i+j-1) + x(i) * y(j)
 20      continue
 30   continue
      return
      end
```

```
C     The gateway routine.
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
      integer nlhs, nrhs
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC Alpha
C     and the SGI 64-bit platforms
C
      integer plhs(*), prhs(*)
      integer mxGetPr, mxGetPi, mxCreateFull
C-----------------------------------------------------------------
C
      integer mx, nx, my, ny, nz
      integer mxGetM, mxGetN, mxIsComplex
      complex*16 x(100), y(100), z(199)

C     Check for proper number of arguments.
      if (nrhs .ne. 2) then
         call mexErrMsgTxt('Two inputs required.')
      elseif (nlhs .gt. 1) then
         call mexErrMsgTxt('Too many output arguments.')
      endif

C     Check that inputs are both row vectors.
      mx = mxGetM(prhs(1))
      nx = mxGetN(prhs(1))
      my = mxGetM(prhs(2))
      ny = mxGetN(prhs(2))
      nz = nx+ny-1

C     Only handle row vector input.
      if(mx .ne. 1 .or. my .ne. 1) then
         call mexErrMsgTxt('Both inputs must be row vector.')
C     Check sizes of the two input.
      elseif(nx .gt. 100 .or. ny .gt. 100) then
         call mexErrMsgTxt('Inputs must have less than 100
                           elements.')
```

```
C     Check to see both inputs are complex.
      elseif ((mxIsComplex(prhs(1)) .ne. 1) .or. +
              (mxIsComplex(prhs(2)) .ne. 1)) then
         call mexErrMsgTxt('Inputs must be complex.')
      endif

C     Create the output array.
      plhs(1) = mxCreateFull(1, nz, 1)

C     Load the data into Fortran arrays(native COMPLEX data).
      call mxCopyPtrToComplex16(mxGetPr(prhs(1)),
                                mxGetPi(prhs(1)), x, nx)
      call mxCopyPtrToComplex16(mxGetPr(prhs(2)),
                                mxGetPi(prhs(2)), y, ny)

C     Call the computational subroutine.
      call convec(x, y, z, nx, ny)

C     Load the output into a MATLAB array.
      call mxCopyComplex16ToPtr(z,mxGetPr(plhs(1)),
                                mxGetPi(plhs(1)), nz)

      return
      end
```

Entering these numbers at the MATLAB prompt

```
x = [3 - 1i, 4 + 2i, 7 - 3i]

x =

   3.0000 - 1.0000i   4.0000 + 2.0000i   7.0000 - 3.0000i

y = [8 - 6i, 12 + 16i, 40 - 42i]

y =

   8.0000 - 6.0000i  12.0000 +16.0000i  40.0000 -42.0000i
```

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 the built-in MATLAB function `conv.m` produces.

Dynamic Allocation of Memory

It is possible to allocate memory dynamically in a Fortran MEX-file, but you must use `%val` to do it. This example takes an input matrix of real data and doubles each of its elements.

```
C     $Revision: 1.5 $
C================================================================
C
C     dblmat.f
C     Example for illustrating how to use %val.
C    Doubles the input matrix. The demo only handles real part C
of input.
C     NOTE: If your Fortran compiler does not support %val,
C     use mxCopy_routine.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-1998 The MathWorks, Inc.
C================================================================
C
```

```
C     Computational subroutine
      subroutine dbl_mat(out_mat, in_mat, size)
      integer size, i
      real*8  out_mat(*), in_mat(*)

      do 10 i=1,size
         out_mat(i) = 2*in_mat(i)
 10   continue

      return
      end

C     Gateway subroutine

      subroutine mexfunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC Alpha
C     and the SGI 64-bit platforms.
C
      integer plhs(*), prhs(*)
      integer pr_in, pr_out
      integer mxGetPr, mxCreateFull
C-----------------------------------------------------------------
C
      integer nlhs, nrhs, mxGetM, mxGetN
      integer m_in, n_in, size

      if(nrhs .ne. 1) then
         call mexErrMsgTxt('One input required.')
      endif
      if(nlhs .gt. 1) then
         call mexErrMsgTxt('Less than one output required.')
      endif
```

```
      m_in = mxGetM(prhs(1))
      n_in = mxGetN(prhs(1))
      size = m_in * n_in
      pr_in = mxGetPr(prhs(1))
      plhs(1) = mxCreateFull(m_in, n_in, 0)
      pr_out = mxGetPr(plhs(1))

C     Call the computational routine.
      call dbl_mat(%val(pr_out), %val(pr_in), size)

      return
      end
```

For an input 2-by-3 matrix

```
x = [1 2 3; 4 5 6];
```

typing

```
y = dblmat(x)
```

yields

```
y =
     2     4     6
     8    10    12
```

Handling Sparse Matrices

The MATLAB 5 API provides a set of functions that allow you to create and manipulate sparse matrices from within your MEX-files. 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, refer to "The MATLAB Array" section in Chapter 1 of this book.

Note: Sparse array indexing is zero based, not one based.

This example illustrates how to populate a sparse matrix.

```
C     $Revision: 1.1 $
C=================================================================
C
C     fulltosparse.f
C     Example for illustrating how to populate a sparse matrix.
C     For the purpose of this example, you must pass in a
C     non-sparse 2-dimensional argument of type real double.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-98 by The MathWorks, Inc.
C=================================================================
C     Load sparse data subroutine.
      function loadsparse(a, b, ir, jc, m, n, nzmax)
      integer nzmax, m, n
      integer ir(*), jc(*)
      real*8 a(*), b(*)

      integer i, j, k

C     Copy nonzeros.
      k = 1
      do 100 j=1,n
C        NOTE: Sparse indexing is zero based.
         jc(j) = k-1
         do 200 i=1,m
            if (a((j-1)*m+i).ne. 0.0) then
               if (k .gt. nzmax) then
                  jc(n+1) = nzmax
                  loadsparse = 1
                  goto 300
               endif
               b(k) = a((j-1)*m+i)
C              NOTE: Sparse indexing is zero based.
               ir(k) = i-1
               k = k+1
            endif
 200     continue
 100  continue
```

```
C     NOTE: Sparse indexing is zero based.
      jc(n+1) = k-1
      loadsparse = 0
 300  return
      end

C     The gateway routine
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
      integer nlhs, nrhs
C-----------------------------------------------------------------
C    (integer) Replace integer by integer*8 on the DEC Alpha and
C     the SGI 64-bit platforms.
C
      integer plhs(*), prhs(*)
      integer mxGetPr, mxCreateSparse, mxGetIr, mxGetJc
      integer pr, sr, irs, jcs
C-----------------------------------------------------------------
C
      integer m, n, nzmax
      integer mxGetM, mxGetN, mxIsComplex, mxIsDouble
      integer loadsparse

C     Check for proper number of arguments.
      if (nrhs .ne. 1) then
         call mexErrMsgTxt('One input argument required.')
      endif
      if (nlhs .gt. 1) then
         call mexErrMsgTxt('Too many output arguments.')
      endif

C     Check data type of input argument.
      if (mxIsDouble(prhs(1)) .eq. 0) then
       call mexErrMsgTxt('Input argument must be of type double.')
      endif
      if (mxIsComplex(prhs(1)) .eq. 1) then
       call mexErrMsgTxt('Input argument must be real only')
      endif

C     Get the size and pointers to input data.
      m = mxGetM(prhs(1))
```

```
      n = mxGetN(prhs(1))
      pr = mxGetPr(prhs(1))

C     Allocate space.
C     NOTE: Assume at most 20% of the data is sparse.
      nzmax = dble(m*n) *.20 + .5
C     NOTE: The maximum number of non-zero elements cannot be less
C     than the number of columns in the matrix.
      if (n .gt. nzmax) then
         nzmax = n
      endif
      plhs(1) = mxCreateSparse(m, n, nzmax, 0)
      sr = mxGetPr(plhs(1))
      irs = mxGetIr(plhs(1))
      jcs = mxGetJc(plhs(1))

C     Load the sparse data.
      if
     (loadsparse(%val(pr),%val(sr),%val(irs),%val(jcs),m,n,nzmax)
     +     .eq. 1) then
       call mexPrintf('Truncating output, input is > 20%% sparse')
      endif
      return
      end
```

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 MATLAB Functions from Fortran MEX-Files

It's possible to call MATLAB functions, operators, M-files, and even other MEX-files from within your Fortran source code by using the API function `mexCallMATLAB`. This 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.

```
C     $Revision: 1.2 $
C
================================================================
C
C     sincall.f
C
C     Example for illustrating how to use mexCallMATLAB
C
C     Creates an mxArray and passes its associated  pointers (in
C     this demo, only pointer to its real part, pointer to
C     number of rows, pointer to number of columns) to
C     subfunction fill() to get data filled up, then calls
C     mexCallMATLAB to calculate the sin function and plot the
C     result.
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-1998 The MathWorks, Inc.
C
================================================================
```

```
C     Subroutine for filling up data

      subroutine fill(pr, m, n, max)

      real*8  pr(*)
      integer i, m, n, max

      m=max/2
      n=1
      do 10 i=1,m
 10      pr(i)=i*(4*3.1415926/max)

      return
      end

C     Gateway subroutine

      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
      integer nlhs, nrhs
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC Alpha
C     and the SGI 64-bit platforms.
C
      integer plhs(*), prhs(*)
      integer rhs(1), lhs(1)
      integer mxGetPr, mxCreateFull
C-----------------------------------------------------------------
C
      integer m, n, max

C     Initializition
      m=1
      n=1
      max=1000

      rhs(1) = mxCreateFull(max, 1, 0)
```

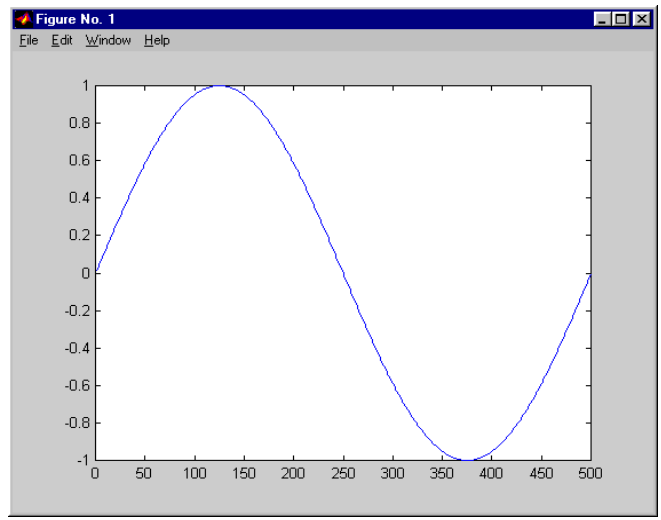
```
C     Pass the integer and variable and let fill() fill up data.
      call fill(%val(mxGetPr(rhs(1))), m, n, max)
      call mxSetM(rhs(1), m)
      call mxSetN(rhs(1), n)

      call mexCallMATLAB(1, lhs, 1, rhs, 'sin')
      call mexCallMATLAB(0, NULL, 1, lhs, 'plot')

C     Cleanup the unfreed memory after calling mexCallMATLAB.
      call mxFreeMatrix(rhs(1))
      call mxFreeMatrix(lhs(1))

      return
      end
```

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, like `eig`, will not work correctly with data that is not double precision.

Running this example

```
sincall
```

displays the results



Note: It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. For example, if you create an M-file that returns two variables but only assigns one of them a value,

```
function [a,b]=foo[c]
a=2*c;
```

you'll get this warning message in MATLAB:

```
Warning: One or more output arguments not assigned during call
to 'foo'.
```

If you then call `foo` using `mexCallMATLAB`, the unassigned output variable will now be of type `mxUNKNOWN_CLASS`.

Advanced Topics

These sections cover advanced features of MEX-files that you can use when your applications require sophisticated MEX-files.

Help Files

Because the MATLAB interpreter chooses the 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 MEX-files. The MATLAB `help` command will automatically find and display the appropriate M-file when help is requested and the interpreter will find and execute the corresponding MEX-file when the function is actually invoked.

Linking Multiple Files

You can combine several source files when building MEX-files. For example,

```
mex circle.f square.o rectangle.f shapes.o
```

is a legal command that operates on the `.f` and `.o` files to create a MEX-file called `circle.`*`ext`*, where *`ext`* is the extension corresponding to the MEX-file type. The name of the resulting MEX-file is taken from the first file in the list.

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 `mex` to combine your object files into a MEX-file. This way you can ensure that your source files are recompiled only when necessary.

Note: On UNIX, you must use the `–fortran` switch to the `mex` script if you are linking Fortran objects.

Variable Scope

Unlike M-file functions, MEX-file functions do not have their own variable workspace. `mexEvalString` evaluates the string in the caller's workspace. In addition, you can use the `mexGetMatrix` and `mexPutMatrix` routines to get and put variables into the caller's workspace.

Memory Management

MATLAB Version 5.2 now implicitly destroys (by calling `mxDestroyArray`) any arrays created by a MEX-file that are not returned in the left-hand side list (`plhs()`). Consequently, any misconstructed arrays left over at the end of a MEX-file's execution have the potential to cause memory errors.

In general, we recommend that MEX-files destroy their own temporary arrays and clean up their own temporary memory. For additional information on memory management techniques, see the "Memory Management" section in Chapter 3 and the "Memory Management Compatibility Issues" section in Chapter 8.

How to Debug Fortran Language MEX-Files

On most platforms, it is now possible to debug MEX-files while they are running within MATLAB. Complete source code debugging, including setting breakpoints, examining variables, and stepping through the source code line-by-line, is now available.

Note: The section, "Troubleshooting," in Chapter 8 provides additional information on isolating problems with MEX-files.

To debug a MEX-file from within MATLAB, you must first compile the MEX-file with the –g option to `mex`.

```
mex –g filename.f
```

Debugging on UNIX

You must start MATLAB from within a debugger. To do this, specify the name of the debugger you want to use with the –D option when starting MATLAB. For example, to use `dbx`, the UNIX debugger, type

```
matlab –Ddbx
```

Once the debugger loads MATLAB into memory, you can start it by issuing a "run" command. Now, from within MATLAB, enable MEX-file debugging by typing

```
dbmex on
```

at the MATLAB prompt. Then run the MEX-file you want to debug as you would ordinarily (either directly or by means of some other function or script). Before executing the MEX-file, you will be returned to the debugger.

You may need to tell the debugger where the MEX-file was loaded or the name of the MEX-file, in which case MATLAB will display the appropriate command for you to use. At this point, you are ready to start debugging. You can list the source code for your MEX-file and set break points in it. It is often convenient

to set one at `mexFunction` so that you stop at the beginning of the gateway function.

Note: The name `mexFunction` may be slightly altered by the compiler (i.e., it may have an underscore appended). To determine how this symbol appears in a given MEX-file, use the UNIX command

```
nm <MEX-file> | grep –i mexfunction
```

To proceed from the breakpoint, issue a “continue” command to the debugger.

Once you hit one of your breakpoints, you can make full use of any facilities your debugger provides to examine variables, display memory, or inspect registers. Refer to the documentation provided with your debugger for information on its use.

If you are at the MATLAB prompt and want to return control to the debugger, you can issue the command

```
dbmex stop
```

which allows you to gain access to the debugger so you can set additional breakpoints or examine source code. To resume execution, issue a “continue” command to the debugger.

Debugging on Windows

DIGITAL Visual Fortran. If you are using the DIGITAL Visual Fortran compiler, you use the Microsoft debugging environment to debug your program.

1. Start the Microsoft Visual Studio by typing at the DOS prompt

   ```
   msdev filename.dll
   ```

2. In the Microsoft environment, from the **Project** menu, select **Settings**. In the window that opens, select the **Debug** tab. This options window contains

edit boxes. In the edit box labeled **Executable for debug session**, enter the full path where MATLAB 5 resides. All other edit boxes should be empty.

3. Open the source files and set a break point on the desired line of code by right-clicking with your mouse on the line of code.

4. From the **Build** menu, select **Debug**, and click **Go**.

5. You will now be able to run your MEX-file in MATLAB and use the Microsoft debugging environment. For more information on how to debug in the Microsoft environment, see the Microsoft Development Studio documentation.

5

Data Export and Import

Using MAT-Files

This section describes the various techniques for importing data to and exporting data from the MATLAB environment. The most important approach involves the use of MAT-files – the data file format that MATLAB uses for saving data to your disk. MAT-files provide a convenient mechanism for moving your MATLAB data between different platforms and for importing and exporting your data to other stand-alone MATLAB applications. To simplify your use of MAT-files in applications outside of MATLAB, 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 routines discussed in the "Creating C Language MEX-Files" and "Creating Fortran MEX-Files" chapters of this book.

This chapter includes these topics:

- Importing and exporting data to and from MATLAB
- Exchanging data files between platforms
- Reading from and writing to MAT-files
- Compiling and linking MAT-file programs

Importing Data to MATLAB

You can introduce data from other programs into MATLAB by several methods. The best method for importing data 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 as an explicit list of elements. If you have a small amount of data, less than 10-15 elements, it is easy to type the data explicitly using brackets [ ]. This method is awkward for larger amounts of data because you can't edit your input if you make a mistake.
- Create data in an M-file. Use your text editor to create an M-file that enters your data as an explicit list of elements. This method is useful when the data isn't already in computer-readable form and you have to type it in. Essentially the same as the first method, this method has the advantage of

allowing you to use your editor to change the data and correct mistakes. You can then just rerun your M-file to re-enter the data.

- Load data from an ASCII flat file. A *flat file* stores the data in ASCII form, with fixed-length rows terminated with new lines (carriage returns) and with spaces separating the numbers. You can edit ASCII flat files using a normal text editor. Flat files can be read directly into MATLAB using the `load` command. The result is to create a variable with the same name as the filename.
- Read data using `fopen`, `fread`, and MATLAB's other low-level I/O functions. This method is useful for loading data files from other applications that have their own established file formats.
- Write a MEX-file to read the data. This is the method of choice if subroutines are already available for reading data files from other applications. See the section, "Introducing MEX-Files," in Chapter 2 for more information.
- Write a program in C or Fortran to translate your data into MAT-file format and then read the MAT-file into MATLAB with the `load` command. Refer to the section, "Reading and Writing MAT-Files," for more information.

Exporting Data from MATLAB

There are several methods for getting MATLAB data back to the outside world:

- For small matrices, use the `diary` command to create a diary file and display the variables, echoing them into this file. You can use your text editor to manipulate the diary file at a later time. The output of `diary` includes the MATLAB commands used during the session, which is useful for inclusion into documents and reports.
- Save the data in ASCII form 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` containing:

  ```
  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 data in numerical matrix form only; numerical arrays (more than 2-dimensions), cell arrays, and structures are not supported.

- Write the data in a special format using fopen, fwrite, and the other low-level I/O functions. This method is useful for writing data files in the file formats required by other applications.
- Develop a MEX-file to write the data. This is the method of choice if subroutines are already available for writing data files in the form needed by other applications. See the section, "Introducing MEX-Files," in Chapter 2 for more information.
- Write out the data as a MAT-file using the save command, and then write a program in C or Fortran to translate the MAT-file into your own special format. See the section, "Reading and Writing MAT-Files," for more information.

Exchanging Data Files Between Platforms

You may want to work with MATLAB implementations on several different computer systems, or need to transmit 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 several different machine architectures requires a facility for exchanging both binary and ASCII data between the various machines. Examples of this type of facility include FTP, NFS, Kermit, and other communication programs. When using these programs, be careful to transmit binary MAT-files in *binary file mode* and ASCII M-files in *ASCII file mode*. Failure to set these modes correctly corrupts the data.

Reading and Writing MAT-Files

The save command in MATLAB saves the MATLAB arrays currently in memory to a binary disk file called a MAT-file. The term MAT-file is used because these files have the extension .mat. The load command performs the

reverse operation: it reads the MATLAB arrays from a MAT-file on disk back into MATLAB's workspace.

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

MAT-File Interface Library

The MAT-file interface library contains a set of subroutines for reading and writing MAT-files. You can call these routines from within your own C and Fortran programs. We recommend that you use these routines, rather than attempt to write your own code, to perform these operations. By using the routines in this library, you will be insulated from future changes to the MAT-file structure.

The MAT-file library contains routines for reading and writing MAT-files. They all begin with the three-letter prefix `mat`. These tables list all the available MAT-functions and their purposes.

Table 5-1: C 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
`matGetFp`	Get an ANSI C file pointer to a MAT-file
`matGetArray`	Read a MATLAB array from a MAT-file
`matPutArray`	Write a MATLAB array to a MAT-file
`matGetNextArray`	Read the next MATLAB array from a MAT-file
`matDeleteArray`	Remove a MATLAB array from a MAT-file

Table 5-1: C MAT-File Routines (Continued)

MAT-Function	Purpose
matPutArrayAsGlobal	Put a MATLAB array into a MAT-file such that the load command will place it into the global workspace
matGetArrayHeader	Load a MATLAB array header from a MAT-file (no data)
matGetNextArrayHeader	Load the next MATLAB array header from a MAT-file (no data)

Table 5-2: 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
matGetMatrix	Get a named MATLAB array from a MAT-file
matPutMatrix	Put a MATLAB array into a MAT-file
matGetNextMatrix	Get the next sequential MATLAB array from a MAT-file
matDeleteMatrix	Remove a MATLAB array from a MAT-file
matGetString	Read a MATLAB string from a MAT-file
matPutString	Write a MATLAB string to a MAT-file

Directory Organization

A collection of files associated with reading and writing MAT-files is located on your disk. Table 5-3 lists the path to the required subdirectories for importing and exporting data using MAT-functions.

Table 5-3: MAT-Function Subdirectories

Platform	Contents	Directories
Windows	Include Files	`<matlab>\extern\include`
	Libraries	`<matlab>\bin`
	Examples	`<matlab>\extern\examples/eng_mat`
UNIX	Include Files	`<matlab>/extern/include`
	Libraries	`<matlab>/extern/lib/$arch`
	Examples	`<matlab>/extern/examples/eng_mat`

The `include` directory holds header files containing function declarations with prototypes for the routines that you can access in the API Library. Included in the subdirectory are:

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

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

Windows

The `bin` subdirectory contains the shared libraries for linking your programs.

Table 5-4: Shared Libraries on Windows

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

UNIX

The `extern/lib/$arch` subdirectory, where `$arch` is your machine's architecture, contains the shared libraries for linking your programs. For example, on `sol2`, the subdirectory is `extern/lib/sol2`.

Table 5-5: Shared Libraries on UNIX

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

so refers to the shared library extension for your platform. For example, on `sol2`, these files are `libmat.so` and `libmx.so`.

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.

Table 5-6: C and Fortran Examples

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

For additional information about the MATLAB API directory organization, see Appendix B, “Directory Organization.”

Examples of MAT-Files

This section includes C and Fortran examples of writing, reading, and diagnosing MAT-files.

Creating a MAT-File

C Example

This sample program illustrates how to use the library routines to create a MAT-file that can be loaded into MATLAB.

```
/* $Revision: 1.2 $ */
/*
 * MAT-file creation program
 *
 * Calling syntax:
 *
 *   matcreat
 *
 * Create a MAT-file that can be loaded into MATLAB.
 *
 * This program demonstrates the use of the following functions:
 *
 *  matClose
 *  matGetArray
 *  matOpen
 *  matPutArray
 *  matPutArrayAsGlobal
 *  Copyright (c) 1984-1998 The MathWorks, Inc.
 */

#include <stdio.h>
#include <stdlib.h>
#include "string.h"
#include "mat.h"
#define BUFSIZE 255
```

```
int create(const char *file) {
  MATFile *pmat;
  mxArray *pa1, *pa2, *pa3;
  double data[9] = { 1.0, 4.0, 7.0, 2.0, 5.0, 8.0, 3.0, 6.0, 9.0 };
  char str[BUFSIZE];

  printf("Creating file %s...\n\n", file);
  pmat = matOpen(file, "w");
  if (pmat == NULL) {
    printf("Error creating file %s\n", file);
   printf("(do you have write permission in this directory?)\n");
    return(1);
  }

  pa1 = mxCreateDoubleMatrix(3,3,mxREAL);
  mxSetName(pa1, "LocalDouble");

  pa2 = mxCreateDoubleMatrix(3,3,mxREAL);
  mxSetName(pa2, "GlobalDouble");
  memcpy((void *)(mxGetData(pa2)), (void *)data,
                 3*3*sizeof(double));

  pa3 = mxCreateString("MATLAB: the language of technical
                       computing");
  mxSetName(pa3, "LocalString");

  matPutArray(pmat, pa1);
  matPutArrayAsGlobal(pmat, pa2);
  matPutArray(pmat, pa3);

  /*
   * Ooops! We need to copy data before writing the array.  (Well,
   * ok, this was really intentional.) This demonstrates that
   * matPutArray will overwrite an existing array in a MAT-file.
   */
  memcpy((char *)(mxGetPr(pa1)), (char *)data,
                 3*3*sizeof(double));
  matPutArray(pmat, pa1);
```

```
/* Clean up. */
mxDestroyArray(pa1);
mxDestroyArray(pa2);
mxDestroyArray(pa3);

if (matClose(pmat) != 0) {
  printf("Error closing file %s\n",file);
  return(1);
}

/*
 * Reopen file and verify its contents with matGetArray.
 */
pmat = matOpen(file, "r");
if (pmat == NULL) {
  printf("Error reopening file %s\n", file);
  return(1);
}

/*
 * Read in each array we just wrote.
 */
pa1 = matGetArray(pmat, "LocalDouble");
if (pa1 == NULL) {
  printf("Error reading existing matrix LocalDouble\n");
  return(1);
}
if (mxGetNumberOfDimensions(pa1) != 2) {
  printf("Error saving matrix: result does not have two
          dimensions\n");
  return(1);
}

pa2 = matGetArray(pmat, "GlobalDouble");
if (pa2 == NULL) {
  printf("Error reading existing matrix LocalDouble\n");
  return(1);
}
```

```
  if (!(mxIsFromGlobalWS(pa2))) {
    printf("Error saving global matrix: result is not global\n");
    return(1);
  }

  pa3 = matGetArray(pmat, "LocalString");
  if (pa3 == NULL) {
    printf("Error reading existing matrix LocalDouble\n");
    return(1);
  }
  mxGetString(pa3, str, BUFSIZE);
 if (strcmp(str, "MATLAB: the language of technical computing"))
{
    printf("Error saving string: result has incorrect
            contents\n");
    return(1);
  }

  /* Clean up before exit. */
  mxDestroyArray(pa1);
  mxDestroyArray(pa2);
  mxDestroyArray(pa3);

  if (matClose(pmat) != 0) {
    printf("Error closing file %s\n",file);
    return(1);
  }
  printf("Done\n");
  return(0);
}

int main()
{
  int result;

  result = create("mattest.mat");
  return (result==0)?EXIT_SUCCESS:EXIT_FAILURE;
}
```

To produce an executable version of this example program, compile the file and link it with the appropriate library. Details on how to compile and link MAT-file programs on the various platforms are discussed in the "Compiling and Linking MAT-File Programs" section.

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, `mattest.mat`, that can be loaded into MATLAB. To run the application, depending on your platform, either double-click on its icon or enter `matcreat` at the system prompt.

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

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

```
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
```

Fortran Example

This example creates a MAT-file, `matdemo.mat`.

```
C $Revision: 1.1 $
C
C     matdemo1.f
C
C     This is a simple program that illustrates how to call the
C     MATLAB MAT-file functions from a Fortran program.  This
C     demonstration focuses on writing MAT-files.
C
C     Copyright (c) 1984-1998 The MathWorks, Inc.
C     All rights reserved
C
```

```
C
C     matdemo1 - Create a new MAT-file from scratch.
C
      program matdemo1
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC alpha
C     and the SGI64 platforms.
C
      integer matOpen, mxCreateFull, mxCreateString
      integer matGetMatrix, mxGetPr
      integer mp, pa1, pa2, pa3
C-----------------------------------------------------------------
C
C     Other variable declarations here
C
      integer status, matClose
      double precision dat(9)
      data dat / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0 /
C
C     Open MAT-file for writing.
C
      write(6,*) 'Creating MAT-file matdemo.mat ...'
      mp = matOpen('matdemo.mat', 'w')
      if (mp .eq. 0) then

  write(6,*) 'Can''t open ''matdemo.mat'' for writing.'
  write(6,*) '(Do you have write permission in this directory?)'

         stop
      end if
C
C     Create variables.
C
      pa1 = mxCreateFull(3,3,0)
      call mxSetName(pa1, 'Numeric')
C
      pa2 = mxCreateString('MATLAB: The language of computing')
      call mxSetName(pa2, 'String')
C
```

```
      pa3 = mxCreateString('MATLAB: The language of computing')
      call mxSetName(pa3, 'String2')
C
      call matPutMatrix(mp, pa1)
      call matPutMatrix(mp, pa2)
      call matPutMatrix(mp, pa3)
C
C     Whoops! Forgot to copy the data into the first matrix --
C     it's now blank. (Well, ok, this was deliberate.)  This
C     demonstrates that matPutMatrix will overwrite existing
C     matrices.
C
      call mxCopyReal8ToPtr(dat, mxGetPr(pa1), 9)
      call matPutMatrix(mp, pa1)
C
C     Now, we'll delete String2 from the MAT-file.
C
      call matDeleteMatrix(mp, 'String2')
C
C     Finally, read back in MAT-file to make sure we know what we
C     put in it.
C
      status = matClose(mp)
      if (status .ne. 0) then
         write(6,*) 'Error closing MAT-file'
         stop
      end if
C
      mp = matOpen('matdemo.mat', 'r')
      if (status .ne. 0) then
         write(6,*) 'Can''t open ''matdemo.mat'' for reading.'
         stop
      end if
C
      pa1 = matGetMatrix(mp, 'Numeric')
      if (mxIsNumeric(pa1) .eq. 0) then
       write(6,*) 'Invalid non-numeric matrix written to MAT-file'
        stop
      end if
```

```
C
      pa2 = matGetMatrix(mp, 'String')
      if (mxIsString(pa2) .eq. 0) then
      write(6,*) 'Invalid non-numeric matrix written to MAT-file'
        stop
      end if
C
      pa3 = matGetMatrix(mp, 'String2')
      if (pa3 .ne. 0) then
         write(6,*) 'String2 not deleted from MAT-file'
         stop
      end if
C
      status = matClose(mp)
      if (status .ne. 0) then
         write(6,*) 'Error closing MAT-file'
         stop
      end if
C
      write(6,*) 'Done creating MAT-file'
      stop
      end
```

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 on 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 MATLAB 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 Windows stand-alone program (not MAT-file specific), see `engwindemo.c` in the `<matlab>\extern\examples\eng_mat` directory.

Reading a MAT-File

C Example

This sample program illustrates how to use the library routines to read and diagnose a MAT-file.

```
/* $Revision: 1.1 $ */
/*
 * MAT-file diagnose program
 *
 * Calling syntax:
 *
 *   matdgns <matfile.mat>
 *
 * It diagnoses the MAT-file named <matfile.mat>.
 *
 * This program demonstrates the use of the following functions:
 *
 *  matClose
 *  matGetDir
 *  matGetNextArray
 *  matGetNextArrayHeader
 *  matOpen
 *
 *  Copyright (c) 1984-1998 The MathWorks, Inc.
 */

#include <stdio.h>
#include <stdlib.h>
#include "string.h"
#include "mat.h"

int diagnose(const char *file) {
  MATFile    *pmat;
  char       **dir;
  int        ndir;
  int        i;
  mxArray *pa;
```

```
printf("Reading file %s...\n\n", file);

/*
 * Open file to get directory.
 */
pmat = matOpen(file, "r");
if (pmat == NULL) {
  printf("Error opening file %s\n", file);
  return(1);
}

/*
 * Get directory of MAT-file.
 */
dir = matGetDir(pmat, &ndir);
if (dir == NULL) {
  printf("Error reading directory of file %s\n", file);
  return(1);
} else {
  printf("Directory of %s:\n", file);
  for (i=0; i < ndir; i++)
    printf("%s\n",dir[i]);
}
mxFree(dir);

/* In order to use matGetNextXXX correctly, reopen file to read
   in headers. */
if (matClose(pmat) != 0) {
  printf("Error closing file %s\n",file);
  return(1);
}
pmat = matOpen(file, "r");
if (pmat == NULL) {
  printf("Error reopening file %s\n", file);
  return(1);
}
```

```
/* Get headers of all variables. */
printf("\nExamining the header for each variable:\n");
for (i=0; i < ndir; i++) {
  pa = matGetNextArrayHeader(pmat);
  if (pa == NULL) {
     printf("Error reading in file %s\n", file);
     return(1);
  }
  /* Diagnose header pa. */
  printf("According to its header, array %s has %d
          dimensions\n", mxGetName(pa),
          mxGetNumberOfDimensions(pa));
  if (mxIsFromGlobalWS(pa))
    printf("  and was a global variable when saved\n");
  else
    printf("  and was a local variable when saved\n");
  mxDestroyArray(pa);
}

/* Reopen file to read in actual arrays. */
if (matClose(pmat) != 0) {
  printf("Error closing file %s\n",file);
  return(1);
}
pmat = matOpen(file, "r");
if (pmat == NULL) {
  printf("Error reopening file %s\n", file);
  return(1);
}

/* Read in each array. */
printf("\nReading in the actual array contents:\n");
for (i=0; i<ndir; i++) {
    pa = matGetNextArray(pmat);
    if (pa == NULL) {
       printf("Error reading in file %s\n", file);
       return(1);
    }
```

```
        /*
         * Diagnose array pa.
         */
        printf("According to its contents, array %s has %d
                dimensions\n", mxGetName(pa),
                mxGetNumberOfDimensions(pa));
        if (mxIsFromGlobalWS(pa))
         printf("  and was a global variable when saved\n");
        else
         printf("  and was a local variable when saved\n");
        mxDestroyArray(pa);
    }

    if (matClose(pmat) != 0) {
        printf("Error closing file %s\n",file);
        return(1);
    }
    printf("Done\n");
    return(0);
}

int main(int argc, char **argv)
{

    int result;

    if (argc > 1)
      result = diagnose(argv[1]);
    else{
      result = 0;
      printf("Usage: matdgns <matfile>");
      printf("where <matfile> is the name of the MAT-file");
      printf("to be diagnosed");
    }

    return (result==0)?EXIT_SUCCESS:EXIT_FAILURE;

}
```

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
```

Fortran Example

This sample program illustrates how to use the library routines to read in the MAT-file created by `matdemo1.f` and describe its contents.

```
C     matdemo2.f
C
C     This is a simple program that illustrates how to call the
C     MATLAB MAT-file functions from a Fortran program.  This
C     demonstration focuses on reading MAT-files. It reads in
C     the MAT-file created by matdemo1.f and describes its
C     contents.
C
C     Copyright (c) 1996-1998 The MathWorks, Inc.
C     All rights reserved
C-----------------------------------------------------------------
C     $Revision: 1.4 $
C
      program matdemo2
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC alpha
C     and the SGI64 platforms.
C
      integer matOpen, matGetDir, matGetNextMatrix
      integer mp, dir, adir(100), pa
C-----------------------------------------------------------------
C
C     Other variable declarations here
C
      integer   mxGetM, mxGetN, matClose
      integer   ndir, i, stat
      character*32 names(100), name, mxGetName
C
C-----------------------------------------------------------------
C     Open file and read directory.
C-----------------------------------------------------------------
```

```
C
      mp = matOpen('matdemo.mat', 'r')
      if (mp .eq. 0) then
         write(6,*) 'Can''t open ''matdemo.mat''.'
         stop
      end if
C
C     Read directory.
C
      dir = matgetdir(mp, ndir)
      if (dir .eq. 0) then
         write(6,*) 'Can''t read directory.'
         stop
      endif
C
C     Copy integer into an array of pointers.
C
      call mxCopyPtrToPtrArray(dir, adir, ndir)
C
C     Copy integer to character string
C
      do 20 i=1,ndir
         call mxCopyPtrToCharacter(adir(i), names(i), 32)
   20 continue
C
      write(6,*) 'Directory of Mat-file:'
      do 30 i=1,ndir
         write(6,*) names(i)
   30 continue
C
      stat = matClose(mp)
      if (stat .ne. 0) then
         write(6,*) 'Error closing ''matdemo.mat''.'
         stop
      end if
```

```
C
C-------------------------------------------------------------
C     Reopen file and read full arrays.
C-------------------------------------------------------------
C
      mp = matOpen('matdemo.mat', 'r')
      if (mp .eq. 0) then
         write(6,*) 'Can''t open ''matdemo.mat''.'
         stop
      end if
C
C     Read directory.
C
      write(6,*) 'Getting full array contents:'
      pa = matGetNextMatrix(mp)
      do while (pa .ne. 0)
C
C     Copy name to character string.
C
         name = mxGetName(pa)
         write(6,*) 'Retrieved ', name
        write(6,*) '  With size ', mxGetM(pa), '-by-', mxGetN(pa)
         pa = matGetNextMatrix(mp)
      end do
C
      stat = matClose(mp)
      if (stat .ne. 0) then
         write(6,*) 'Error closing ''matdemo.mat''.'
         stop
      end if
      stop
C
      end
```

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

This section describes the steps required to compile and link MAT-file programs on UNIX and Windows systems. It begins by looking at a special consideration for compilers that do not mask floating-point exceptions.

Special Considerations

Floating-Point Exceptions

Certain mathematical operations can result in nonfinite values. For example, division by zero results in the nonfinite IEEE value, `inf`. A floating-point exception occurs when such an operation is performed. Because MATLAB uses an IEEE model that supports nonfinite values such as `inf` and `NaN`, MATLAB disables, or *masks*, floating-point exceptions.

Some compilers do not mask floating-point exceptions by default. This causes MAT-file applications built with such compilers to terminate when a floating-point exception occurs. Consequently, you need to take special precautions when using these compilers to mask floating-point exceptions so that your MAT-file application will perform properly.

Note: MATLAB-based applications should never get floating-point exceptions. If you do get a floating-point exception, verify that any third party libraries that you link against do not enable floating-point exception handling.

This table shows the platforms and compilers on which you should mask floating-point exceptions.

Platform	Compiler
DEC Alpha	DIGITAL Fortran 77
Linux	Absoft Fortran
Windows	Borland C++

DEC Alpha. To mask floating-point exceptions on the DEC Alpha platform, use the –fpe3 compile flag. For example,

```
f77 –fpe3
```

Absoft Fortran Compiler on Linux. To mask floating-point exceptions when using the Absoft Fortran compiler on the Linux platform, you must add some code to your program. Include the following at the beginning of your `main()` program, before any calls to MATLAB API functions.

```
      integer cw, arm387
C
      cw = arm387(z'0000003F')
      cw = cw .or. z'0000003F'
      call arm387(cw)
```

Borland C++ Compiler on Windows. To mask floating-point exceptions when using the Borland C++ compiler on the Windows platform, you must add some code to your program. Include the following at the beginning of your `main()` or `WinMain()` function, before any calls to MATLAB API functions.

```
#include <float.h>
    .
    .
    .
_control87(MCW_EM,MCW_EM);
    .
    .
    .
```

UNIX

Under UNIX at runtime, you must tell the system where the API shared libraries reside. These sections provide the necessary UNIX commands depending on your shell and system architecture.

Setting Runtime Library Path

In C shell, the command to set the library path is

```
setenv LD_LIBRARY_PATH <matlab>/extern/lib/$Arch:$LD_LIBRARY_PATH
```

In Bourne shell, the commands to set the library path are

```
LD_LIBRARY_PATH=<matlab>/extern/lib/$Arch:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

where:

<matlab> is the MATLAB root directory and $Arch is your system architecture (alpha, lnx86, sgi, sol2, sun4, hp700, ibm_rs, or sgi64). Note that the environment variable (LD_LIBRARY_PATH in this example) varies on several platforms. Table 5-7 lists the different environment variable names you should use on these systems.

Table 5-7: Environment Variables Name

Architecture	Environment Variable
HP700	SHLIB_PATH
IBM RS/6000	LIBPATH
SGI 64	LD_LIBRARY64_PATH

It is convenient to place these commands in a startup script such as ~/.cshrc for C shell or ~/.profile for Bourne shell.

Compiling and Linking Commands

Table 5-8 provides the commands required to compile and link MAT-file C and Fortran programs on UNIX platforms.

Table 5-8: Compiling and Linking MAT-File Programs

UNIX – C	Command
HP700	cc –Aa <include dir> –o <result> <source> <libdir> <libraries>
SGI 64	cc –64 –mips4 <include dir> –o <result> <source> <libdir> <libraries>
Linux	gcc –ansi <include dir> –o <result> <source> <libdir> <libraries>
SunOS 4.x	acc <include dir> –o <result> <source> <libdir> <libraries>
All others	cc <include dir> –o <result> <source> <libdir> <libraries>

Table 5-8: Compiling and Linking MAT-File Programs (Continued)

UNIX – Fortran	
SGI 64	`f77 –64 –mips4 <include dir> –o <result> <source> <libdir> <libraries>`
DEC Alpha	`f77 –fpe3 <include dir> –o <result> <source> <libdir> <libraries>`
All others	`f77 <include dir> –o <result> <source> <libdir> <libraries>`

	where:
	`<include dir>` is `–I<matlab>/extern/include`
	`<result>` is the name of the resulting program
	`<source>` is the list of source files
	`<libdir>` is `–L<matlab>/extern/lib/$ARCH`
	`<libraries>` is `–lmat –lmx`

Special Consideration for Fortran (f77) on HP-UX 10.x

In the version of the Fortran compiler (`f77`) that ships with HP-UX 10.x, the meaning of the `–L` flag has changed. The `–L` flag now requests a listing. So, to force the linker to see the libraries you want linked in with your code, use

```
f77 <include dir> –o <result> <source> –Wl,–L<matlab>/extern/lib/hp700 <libraries>
```

Windows

To compile and link Fortran or C MAT-file programs, use the `mex` script with a MAT options file. The file, `df50engmatopts.bat` (DIGITAL Visual Fortran), is for stand-alone Fortran MAT programs. The files, `watengmatopts.bat`, `wat11engmatopts.bat`, `bccengmatopts.bat`, `msvc50engmatopts.bat`, and `msvcengmatopts.bat` are for stand-alone C MAT programs. You can find all of these files in `<matlab>\bin`. Refer to them for details on how to customize a MAT options file for your particular compiler.

As an example, to compile and link a stand-alone MAT application on Windows using MSVC (Version 5.0) use

```
mex –f <matlab>\bin\msvc50engmatopts.bat filename.c
```

where *`filename`* is the name of the source file.

6

Using the MATLAB Engine

Interprocess Communication: The MATLAB Engine

The MATLAB engine library is a set of routines that allows you to call MATLAB from your own programs, thereby employing MATLAB as a computation engine. 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, the MATLAB engine can run on your machine, or on any other UNIX machine on your network, including machines of a different architecture. Thus you could implement a user interface on your workstation and perform the computations on a faster machine located elsewhere on your network. See the `engOpen` reference page, which is accessible from the MATLAB Help Desk, for further information.
- Instead of requiring that all of MATLAB be linked to your program (a substantial amount of code), only a small engine communication library is needed.

The Engine Library

The engine library contains the following routines for controlling the MATLAB computation engine. Their names all begin with the three-letter prefix `eng`. These tables list all the available engine functions and their purposes.

Table 6-1: C Engine Routines

Function	Purpose
`engOpen`	Start up MATLAB engine
`engClose`	Shut down MATLAB engine
`engGetArray`	Get a MATLAB array from the MATLAB engine
`engPutArray`	Send a MATLAB array to the MATLAB engine
`engEvalString`	Execute a MATLAB command
`engOutputBuffer`	Create a buffer to store MATLAB text output

Table 6-2: Fortran Engine Routines

Function	Purpose
`engOpen`	Start up MATLAB engine
`engClose`	Shut down MATLAB engine
`engGetMatrix`	Get a MATLAB array from the MATLAB engine
`engPutMatrix`	Send a MATLAB array to the MATLAB engine
`engEvalString`	Execute a MATLAB command
`engOutputBuffer`	Create a buffer to store MATLAB text output

The MATLAB engine also uses the `mx` prefixed API routines discussed in the "Creating C Language MEX-Files" and "Creating Fortran MEX-Files" chapters of this book.

Communicating with MATLAB

On UNIX, the engine library communicates with the MATLAB engine using pipes, and, if needed, `rsh` for remote execution. On Microsoft Windows, the engine library communicates with MATLAB using ActiveX. The next chapter, "Client/Server Applications," contains a detailed description of ActiveX.

Note: On the PC, support for MATLAB 5 data types and sparse matrices is not available in engine applications.

Examples

Calling the MATLAB Engine

C Example

This program, `engdemo.c`, illustrates how to call the engine functions from a stand-alone C program. For the Windows version of this program, see `engwindemo.c` in the `<matlab>\extern\examples\eng_mat` directory. Engine examples, like the MAT-file examples, are located in the `eng_mat` directory.

```
/* $Revision: 1.3 $ */
/*
 * engdemo.c
 *
 * This is a simple program that illustrates how to call the
 * MATLAB engine functions from a C program.
 *
 *      Copyright (c) 1996-1998 The MathWorks, Inc.
 *      All rights reserved
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "engine.h"
#define  BUFSIZE 256

int main()

{
   Engine *ep;
   mxArray *T = NULL, *result = NULL;
   char buffer[BUFSIZE];
   double time[10] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0,
                       8.0, 9.0 };
```

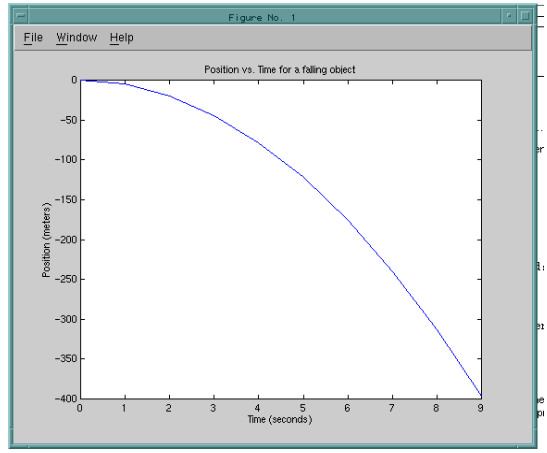
```
/*
 * Start the MATLAB engine locally by executing the string
 * "matlab".
 *
 * To start the session on a remote host, use the name of
 * the host as the string rather than \0
 *
 * For more complicated cases, use any string with whitespace,
 * and that string will be executed literally to start MATLAB.
 */
if (!(ep = engOpen("\0"))) {
   fprintf(stderr, "\nCan't start MATLAB engine\n");
   return EXIT_FAILURE;
}

/*
 * PART I
 *
 * For the first half of this demonstration, we will send data
 * to MATLAB, analyze the data, and plot the result.
 */

/*
 * Create a variable for our data.
 */
T = mxCreateDoubleMatrix(1, 10, mxREAL);
mxSetName(T, "T");
memcpy((void *)mxGetPr(T), (void *)time, sizeof(time));
/*
 * Place the variable T into the MATLAB workspace.
 */
engPutArray(ep, T);

/*
 * Evaluate a function of time, distance = (1/2)g.*t.^2
 * (g is the acceleration due to gravity).
 */
engEvalString(ep, "D = .5.*(-9.8).*T.^2;");
```

```
/*
 * Plot the result.
 */
engEvalString(ep, "plot(T,D);");
engEvalString(ep, "title('Position vs. Time for a falling
                          object');");
engEvalString(ep, "xlabel('Time (seconds)');");
engEvalString(ep, "ylabel('Position (meters)');");

/*
 * Use fgetc() to make sure that we pause long enough to be
 * able to see the plot.
 */
printf("Hit return to continue\n\n");
fgetc(stdin);
/*
 * We're done for Part I! Free memory, close MATLAB engine.
 */
printf("Done for Part I.\n");
mxDestroyArray(T);
engEvalString(ep, "close;");
/*
 * PART II
 *
 * For the second half of this demonstration, we will request
 * a MATLAB string, which should define a variable X.  MATLAB
 * will evaluate the string and create the variable.  We
 * will then recover the variable, and determine its type.
 */

/*
 * Use engOutputBuffer to capture MATLAB output, so we can
 * echo it back.
 */

engOutputBuffer(ep, buffer, BUFSIZE);
while (result == NULL) {
    char str[BUFSIZE];
```

```
    /*
     * Get a string input from the user.
     */
    printf("Enter a MATLAB command to evaluate.  This
            command should\n");
    printf("create a variable X.  This program will then
            determine\n");
    printf("what kind of variable you created.\n");
    printf("For example: X = 1:5\n");
    printf(">> ");

    fgets(str, BUFSIZE-1, stdin);

    /*
     * Evaluate input with engEvalString.
     */
    engEvalString(ep, str);
    /*
     * Echo the output from the command.  First two characters
     * are always the double prompt (>>).
     */
    printf("%s", buffer+2);

    /*
     * Get result of computation.
     */
    printf("\nRetrieving X...\n");
    if ((result = engGetArray(ep,"X")) == NULL)
      printf("Oops! You didn't create a variable X.\n\n");
    else {
   printf("X is class %s\t\n", mxGetClassName(result));
    }
}
```

```
    /*
     * We're done! Free memory, close MATLAB engine and exit.
     */
    printf("Done!\n");
    mxDestroyArray(result);
    engClose(ep);

    return EXIT_SUCCESS;
}
```

The first part of this program launches MATLAB and sends it data. MATLAB then analyzes the data and plots the results.



The program then continues with

```
Hit 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.

Fortran Example

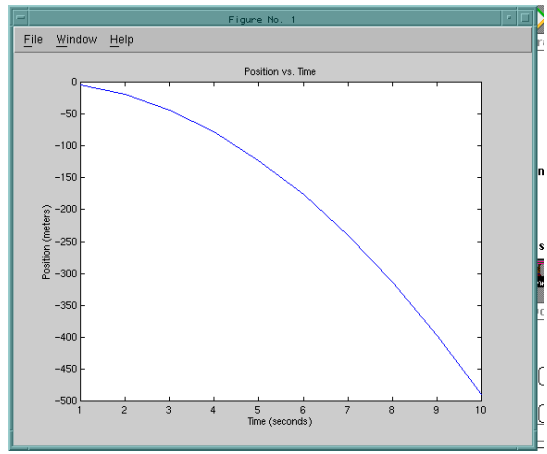
This program, `fengdemo.f`, illustrates how to call the engine functions from a stand-alone Fortran program.

```
C
C     fengdemo.f
C
C     This program illustrates how to call the
C     MATLAB Engine functions from a Fortran program.
C
C     Copyright (c) 1997-1998 The MathWorks, Inc.
C     All rights reserved
C=================================================================
C $Revision: 1.2 $

      program main
C-----------------------------------------------------------------
C     (integer) Replace integer by integer*8 on the DEC Alpha
C     and the SGI 64 platforms.
C
      integer engOpen, engGetMatrix, mxCreateFull, mxGetPr
      integer ep, T, D, result
C-----------------------------------------------------------------
```

```
C
C    Other variable declarations here
     double precision time(10), dist(10)
     integer stat, temp
    data time / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0 /
C
      ep = engOpen('matlab ')
C
      if (ep .eq. 0) then
         write(6,*) 'Can''t start MATLAB engine'
         stop
      endif
C
      T = mxCreateFull(1, 10, 0)
      call mxSetName(T, 'T')
      call mxCopyReal8ToPtr(time, mxGetPr(T), 10)
C
C
C     Place the variable T into the MATLAB workspace.
C
      call engPutMatrix(ep, T)
C
C
C     Evaluate a function of time, distance = (1/2)g.*t.^2
C     (g is the acceleration due to gravity).
C
      call engEvalString(ep, 'D = .5.*(-9.8).*T.^2;')
C
C
C     Plot the result.
C
      call engEvalString(ep, 'plot(T,D);')
      call engEvalString(ep, 'title(''Position vs. Time'')')
      call engEvalString(ep, 'xlabel(''Time (seconds)'')')
      call engEvalString(ep, 'ylabel(''Position (meters)'')')
C
```

```
C
C    Read from console to make sure that we pause long enough to be
C     able to see the plot.
C
      print *, 'Type 0 <return> to Exit'
      print *, 'Type 1 <return> to continue'

      read(*,*) temp

      if (temp.eq.0) then
         print *, 'EXIT!'
         stop
      end if
C
      call engEvalString(ep, 'close;')
C
      D = engGetMatrix(ep, 'D')
      call mxCopyPtrToReal8(mxGetPr(D), dist, 10)
      print *, 'MATLAB computed the following distances:'
      print *, '  time(s)  distance(m)'
      do 10 i=1,10
         print 20, time(i), dist(i)
 20      format(' ', G10.3, G10.3)
 10   continue
C
C
      call mxFreeMatrix(T)
      call mxFreeMatrix(result)
      stat = engClose(ep)
C
      stop
      end
```

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.

Compiling and Linking Engine Programs

To produce an executable version of an engine program, you must compile it and link it with the appropriate library. This section describes the steps required to compile and link engine programs on UNIX and Windows systems. It begins by looking at a special consideration for compilers that do not mask floating-point exceptions.

Special Considerations

Floating-Point Exceptions

Certain mathematical operations can result in nonfinite values. For example, division by zero results in the nonfinite IEEE value, `inf`. A floating-point exception occurs when such an operation is performed. Because MATLAB uses an IEEE model that supports nonfinite values such as `inf` and `NaN`, MATLAB disables, or *masks*, floating-point exceptions.

Some compilers do not mask floating-point exceptions by default. This causes engine programs built with such compilers to terminate when a floating-point exception occurs. Consequently, you need to take special precautions when using these compilers to mask floating-point exceptions so that your engine application will perform properly.

Note: MATLAB-based applications should never get floating-point exceptions. If you do get a floating-point exception, verify that any third party libraries that you link against do not enable floating-point exception handling.

This table shows the platforms and compilers on which you should mask floating-point exceptions.

Platform	Compiler
DEC Alpha	DIGITAL Fortran 77
Linux	Absoft Fortran
Windows	Borland C++

DEC Alpha. To mask floating-point exceptions on the DEC Alpha platform, use the –fpe3 compile flag. For example,

```
f77 –fpe3
```

Absoft Fortran Compiler on Linux. To mask floating-point exceptions when using the Absoft Fortran compiler on the Linux platform, you must add some code to your program. Include the following at the beginning of your main() program, before any calls to MATLAB API functions.

```
      integer cw, arm387
C
      cw = arm387(z'0000003F')
      cw = cw .or. z'0000003F'
      call arm387(cw)
```

Borland C++ Compiler on Windows. To mask floating-point exceptions when using the Borland C++ compiler on the Windows platform, you must add some code to your program. Include the following at the beginning of your main() or WinMain() function, before any calls to MATLAB API functions.

```
#include <float.h>
    .
    .
    .
_control87(MCW_EM,MCW_EM);
    .
    .
    .
```

UNIX

Under UNIX at runtime, you must tell the system where the API shared libraries reside. These sections provide the necessary UNIX commands depending on your shell and system architecture.

Setting Runtime Library Path

In C shell, the command to set the library path is

```
setenv LD_LIBRARY_PATH <matlab>/extern/lib/$Arch:$LD_LIBRARY_PATH
```

In Bourne shell, the commands to set the library path are

```
LD_LIBRARY_PATH=<matlab>/extern/lib/$Arch:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

where:

`<matlab>` is the MATLAB root directory and `$Arch` is your system architecture (`alpha`, `lnx86`, `sgi`, `sol2`, `sun4`, `hp700`, `ibm_rs`, or `sgi64`). Note that the environment variable (`LD_LIBRARY_PATH` in this example) varies on several platforms. Table 6-3 lists the different environment variable names you should use on these systems.

Table 6-3: Environment Variables Name

Architecture	Environment Variable
HP700	`SHLIB_PATH`
IBM RS/6000	`LIBPATH`
SGI 64	`LD_LIBRARY64_PATH`

It is convenient to place these commands in a startup script such as `~/.cshrc` for C shell or `~/.profile` for Bourne shell.

Compiling and Linking Commands

Table 6-4 provides the commands required to compile and link engine C and Fortran programs on UNIX platforms.

Table 6-4: Compiling and Linking Engine Programs

UNIX – C	Command
HP700	`cc –Aa <include dir> –o <result> <source> <libdir> <libraries>`
SGI 64	`cc –64 –mips4 <include dir> –o <result> <source> <libdir> <libraries>`
Linux	`gcc –ansi <include dir> –o <result> <source> <libdir> <libraries>`
SunOS 4.x	`acc <include dir> –o <result> <source> <libdir> <libraries>`
All others	`cc <include dir> –o <result> <source> <libdir> <libraries>`
UNIX – Fortran	
SGI 64	`f77 –64 –mips4 <include dir> –o <result> <source> <libdir> <libraries>`
DEC Alpha	`f77 –fpe3 <include dir> –o <result> <source> <libdir> <libraries>`
All others	`f77 <include dir> –o <result> <source> <libdir> <libraries>`

	where:
	`<include dir>` is `–I<matlab>/extern/include`
	`<result>` is the name of the resulting program
	`<source>` is the list of source files
	`<libdir>` is `–L<matlab>/extern/lib/$ARCH`
	`<libraries>` is `–leng –lmx`

You can now run the executable you have just produced.

Special Consideration for Fortran (f77) on HP-UX 10.x

In the version of the Fortran compiler (`f77`) that ships with HP-UX 10.x, the meaning of the `–L` flag has changed. The `–L` flag now requests a listing. So, to force the linker to see the libraries you want linked in with your code, use

```
f77 <include dir> –o <result> <source> –Wl,–L<matlab>/extern/lib/hp700 <libraries>
```

Windows

To compile and link engine programs, use the `mex` script with an engine options file. `watengmatopts.bat`, `wat11engmatopts.bat`, `bccengmatopts.bat`, `df50engmatopts.bat`, `msvc50engmatopts.bat`, and `msvcengmatopts.bat` are stand-alone engine and MAT options files (located in `<matlab>\bin`). For example, to compile and link a stand-alone engine application on Windows using MSVC (Version 5.0) use

```
mex –f <matlab>\bin\msvc50engmatopts.bat filename.c
```

where *`filename`* is the name of the source file.

7

Client/Server Applications

MATLAB ActiveX Integration

What Is ActiveX?

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

ActiveX Concepts and Terminology

COM. ActiveX is a family of related object-oriented technologies that have a common root, called the *Component Object Model*, or COM. Each object-oriented language or environment has an object model that defines certain characteristics of objects in that environment, such as how objects are located, instantiated, or identified. COM defines the object model for all ActiveX objects.

ActiveX Interfaces. Each ActiveX object supports one or more named interfaces. An *interface* is a logically related collection of methods, properties, and events. Methods are similar to function calls in that they are a request for the object to perform some action. Properties are state variables maintained by the object, such as the color of text, or the name of a file on which the control is acting. Events are notifications that the control forwards back to its client (similar to Handle Graphics® callbacks.) For example, the sample control shipped with MATLAB has the following methods, properties, and events:

Methods

`Redraw` - causes the control to redraw
`Beep` - causes the control to beep
`AboutBox` - display the control's "About" dialog

Properties

`Radius` (integer) - sets the radius of the circle drawn by the control
`Label` (string) - text to be drawn in the control

Events

`Click` - fired when the user clicks on the control

One important characteristic of COM is that it defines an object model in which objects support multiple interfaces. Some interfaces are *standard* interfaces, which are defined by Microsoft and are part of ActiveX, and some interfaces are *custom* interfaces, which are defined by individual component vendors. In order to use any ActiveX object, you must learn about which custom interfaces it supports, and the interface's methods, properties, and events. The ActiveX object's vendor provides this information.

MATLAB ActiveX Support Overview

MATLAB supports two ActiveX technologies: ActiveX control containment and ActiveX Automation. ActiveX controls are application components that can be both visually and programmatically integrated into an ActiveX control container, such as MATLAB figure windows. Some examples of useful ActiveX controls are the Microsoft Internet Explorer Web Browser control, the Microsoft Windows Communications control for serial port access, and the graphical user interface controls delivered with the Visual Basic development environment.

ActiveX Automation allows MATLAB to both control and be controlled by other ActiveX components. When MATLAB is controlled by another component, it is acting as an automation *server*. When MATLAB controls another component, MATLAB is the automation *client*, and the other component is the automation *server*.

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. MATLAB automation client capabilities allow MATLAB, through M-code, to programmatically instantiate and manipulate automation servers. The MATLAB automation client capabilities are a subset of the MATLAB control containment support, since you use the automation client capabilities to manipulate controls as well as automation servers. In other words, all ActiveX controls are ActiveX automation servers, but not all automation servers are necessarily controls.

In general, servers that are not controls will not be physically or visually embedded in the client application. (MATLAB is a good example — MATLAB is not itself a control, but it is a server. So, MATLAB cannot be physically

embedded within another client. However, since MATLAB is a control *container*, other ActiveX controls can be embedded within MATLAB.)

In addition, MATLAB ships with a very simple sample ActiveX control that draws a circle on the screen and displays some text. This allows MATLAB users to try out MATLAB's ActiveX control support with a known control. For more information, see the section, "MATLAB Sample Control."

MATLAB ActiveX Client Support

In order to use an ActiveX component with MATLAB or with any ActiveX client, you first need to consult the documentation for that object and find out the name of the object itself (known as the "ProgID"), as well as the names of the interfaces, methods, properties, and events that the object uses. Once you have this information, you can integrate that object with MATLAB by using the ActiveX client support.

Using ActiveX Objects

You create an ActiveX control or server in MATLAB by creating an instance of the MATLAB activex class. Each instance represents one interface to the object.

Note: This book uses ActiveX to refer to the generic ActiveX control/server and activex to refer to the MATLAB class/object.

Creating ActiveX Objects. There are two commands used to create activex objects initially

`actxcontrol`	Creates an ActiveX control.
`actxserver`	Creates an ActiveX automation server.

Once you create an activex object that represents an interface, you can manipulate it by invoking methods on the object to perform various actions.

Manipulating the Interface. These methods are implemented for the activex class.

Method	Description
`invoke`	Invokes a method on an interface or displays a list of methods.
`set`	Sets a property on an interface.
`get`	Gets a property value from an interface or displays a list of properties.
`propedit`	Asks the control to display its built-in property page.
`release`	Releases an activex object.
`send`	Displays a list of events.
`delete`	Deletes an activex object.

The creation commands, `actxcontrol` and `actxserver`, both return a MATLAB activex object, which represents the default interface for the object that was created. However, these objects may have other interfaces. It is possible (and common) for interfaces to also be obtained by invoking a method on, or getting a property from, an existing interface. The ActiveX `get` and `invoke` methods automatically create and return new activex objects to represent these additional interfaces.

When each interface is no longer needed, use the `release` method to release the interface. When the entire control or server is no longer needed, use the `delete` command to delete it. See the section, “Releasing Interfaces,” for more details.

ActiveX Client Reference

This section contains the reference pages for the commands that create ActiveX objects and manipulate their interfaces.

actxcontrol

Purpose

Create an ActiveX control in a figure window.

Syntax

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

Arguments

`progid`

String that is the name of the control to create. The control vendor provides this string.

`position`

Position vector containing the `x` and `y` location and the `xsize` and `ysize` of the control, expressed in pixel units as `[x y xsize ysize]`. Defaults to `[20 20 60 60]`.

`handle`

Handle Graphics handle of the figure window in which the control is to be created. If the control should be invisible, use the handle of an invisible figure window. Defaults to `gcf`.

`callback`

Name of an M-function that accepts a variable number of arguments. This function will be called whenever the control triggers an event. Each argument is converted to a MATLAB string; the first argument is always a string that represents the numerical value of the event that was triggered. These numerical values are defined by the control. (See the section, “Writing Event Handlers,” for more information on handling control events.)

`event`

Triggered event specified by either number or name.

`eventhandler`

Name of an M-function that accepts a variable number of arguments. This function will be called whenever the control triggers the event

associated with it. The first argument is the activex object, the second number represents the numerical value of the event that was triggered. Note that the second number is not converted to a string as is the case in the "callback" M-file style. These values are defined by the control. See the section, "Writing Event Handlers," for more information on handling control events.

Note: There are two ways to handle events. You can create a single callback or you can specify a cell array that contains pairs of events and event handlers. In the cell array format, specify events by either number or name. Each control defines event numbers and names. There is no limit to the number of pairs that can be specified in the cell array. Although using the single callback method may be easier in some cases, using the cell array technique creates more efficient code that results in better performance.

Returns

A MATLAB activex object that represents the default interface for this control or server. Use the `get`, `set`, `invoke`, `propedit`, `release`, and `delete` methods on this object. A MATLAB error will be generated if this call fails.

Description

Create an ActiveX control at a particular location within a figure window. If the parent figure window is invisible, the control will be invisible. The returned MATLAB activex object represents the default interface for the control. This interface must be released through a call to `release` when it is no longer needed to free the memory and resources used by the interface. Note that releasing the interface does not delete the control itself (use the `delete` command to delete the control.)

For an example callback event handler, see the file `sampev.m` in the `toolbox\matlab\winfun` directory.

Examples

Callback style:

```
f = figure ('pos', [100 200 200 200]);
% create the control to fill the figure
h = actxcontrol ('MWSAMP.MwsampCtrl.1', [0 0 200 200], ...
                  gcf, 'sampev')
```

Cell array style:

```
h = actxcontrol ('SELECTOR.SelectorCtrl.l', ...
[0 0 200 200], f, {–600 'myclick'; –601 'my2click'; ...
–605 'mymoused'})
```

```
h = actxcontrol ('SELECTOR.SelectorCtrl.l', ...
[0 0 200 200], f, {'Click' 'myclick'; ...
'DblClick' 'my2click'; 'MouseDown' 'mymoused'})
```

where 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: ')
varargin(5)
disp('The Y position is: ')
varargin(6)
```

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

For example

```
h = actxcontrol('SELECTOR.SelectorCtrl.1', ...
[0 0 200 200], f, {'Click' 'allevents'; ...
'DblClick' 'allevents'; 'MouseDown' 'allevents'})
```

and `allevents.m` is

```
function allevents(varargin)
if (varargin{2} = –600)
             disp ('Single Click Event Fired')
elseif (varargin{2} = –601)
             disp ('Double Click Event Fired')
elseif (varargin{2} = –605)
             disp ('Mousedown Event Fired')
end
```

actxserver

Purpose

Create an ActiveX automation server and return an activex object for the server's default interface.

Syntax

```
h = actxserver (progid [, MachineName])
```

Arguments

`progid`

This is a string that is the name of the control to instantiate. This string is provided by the control or server vendor and should be obtained from the vendor's documentation. For example, the `progid` for Microsoft Excel is `Excel.Application`.

`MachineName`

This is the name of a remote machine on which the server is to be run. This argument is optional and is used only in environments that support Distributed Component Object Model (DCOM) — see below. This can be an IP address or a DNS name.

Returns

An activex object that represents the server's default interface. Use the `get`, `set`, `invoke`, `release`, and `delete` methods on this object. A MATLAB error will be generated if this call fails.

Description

Create an ActiveX automation server and return a MATLAB activex object that represents the server's default interface. Local/Remote servers differ from controls in that they are run in a separate address space (and possibly on a separate machine) and are not part of the MATLAB process. Additionally, any user interface that they display will be in a separate window and will not be attached to the MATLAB process. Examples of local servers are Microsoft Excel and Microsoft Word. Note that automation servers do not use callbacks or event handlers.

Example

```
% launches Microsoft Excel & makes main frame window visible
h = actxserver ('Excel.Application')
set (h, 'Visible', 1);
```

invoke

Purpose

Invoke a method on an object's interface and retrieve the return value of the method, if any, or display a list of methods.

Syntax

```
v = invoke (a [, 'methodname' [, arg1, arg2, …]])
```

Arguments

`a`

An activex object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`methodname`

A string that is the name of the method to be invoked.

`arg1, …, argn`

Arguments, if any, required by the method being invoked.

Returns

The value returned by the method or a list of methods (if you use the form `invoke(a)`.) The data type of the value is dependent upon the specific method being invoked and is determined by the specific control or server. If the method returns an interface (described in ActiveX documentation as an *interface*, or an *Idispatch* *), this method will return a new MATLAB activex object that represents the interface returned. See the section, "Data Conversions," for a description of how MATLAB converts ActiveX data types.

Description

Invoke a method on an object's interface and retrieve the return value of the method, if any. (Some methods have no return value.)

Example

Invoke a method:

```
f = figure ('pos', [100 200 200 200]);
% create the control to fill the figure
h = actxcontrol ('MWSAMP.MwsampCtrl.1', [0 0 200 200], f)
set (h, 'Radius', 100);
v = invoke (h, 'Redraw')
```

Display a list of methods:

```
invoke(h)
```

```
AboutBox = Void AboutBox ()
ShowPropertyPage = Void ShowPropertyPage ()
```

set

Purpose

Set an interface property to a specific value.

Syntax

```
set (a [, 'propertyname' [, value [, arg1, arg2, …]]])
```

Arguments

`a`
An activex object handle previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`propertyname`
A string that is the name of the property to be set.

`value`
The value to which the interface property is set.

`arg1, …, argn`
Arguments, if any, required by the property. Properties are similar to methods in that it is possible for a property to have arguments.

Returns

There is no return value from set.

Description

Set an interface property to a specific value. See the section, "Data Conversions," for information on how MATLAB converts workspace matrices to ActiveX data types.

Example

```
f = figure ('pos', [100 200 200 200]);
% create the control to fill the figure
a = actxcontrol ('MWSAMP.MwsampCtrl.1', [0 0 200 200], f)
set (a, 'Label', 'Click to fire event');
set (a, 'Radius', 40);
invoke (a, 'Redraw');
```

get

Purpose

Retrieve a property value from an interface or get a list of properties.

Syntax

```
v = get (a [, 'propertyname' [, arg1, arg2, …]])
```

Arguments

`a`

An activex object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

`propertyname`

A string that is the name of the property value to be retrieved.

`arg1, …, argn`

Arguments, if any, required by the property being retrieved. Properties are similar to methods in that it is possible for a property to have arguments.

Returns

The value of the property or a list of properties (if you use the form `get(a)`). The meaning and type of this value is dependent upon the specific property being retrieved. The object's documentation should describe the specific meaning of the return value. See the section, "Data Conversions," for a description of how MATLAB converts ActiveX data types.

Description

Retrieve a property value from an interface.

Note: You can use the return value of `actxcontrol` as an argument to the `get` function to get a list of properties of the control and methods that can be invoked.

Examples

Retrieve a single, property value:

```
% get the string value of the 'Label' property
s = get (a, 'Label');
```

Retrieve a list of properties:

```
get(a)
```

```
AutoAlign = [-1]
AutoAngle = [-1]
AutoAngleConfine = [0]
AutoOffset = [-1]
.
.
.
SelectionOffsetY = [0]
SelectionRadius = [0.8]
Selections = [4]
Value = [0]
```

propedit

Purpose

Request the control to display its built-in property page.

Syntax

```
propedit (a)
```

Arguments

`a`
An interface handle previously returned from `actxcontrol`, `get`, or `invoke`.

Description

Request the control to display its built-in property page. Note that some controls do not have a built-in property page. For those objects, this command will fail.

Example

```
propedit (a)
```

release

Purpose

Releases an interface.

Syntax

```
release (a)
```

Arguments

a
Activex object that represents the interface to be released.

Description

Release the interface and all resources used by the interface. Each interface handle must be released when you are finished manipulating its properties and invoking its methods. Once an interface has been released, it is no longer valid and subsequent ActiveX operations on the MATLAB object that represents that interface will result in errors.

Note: Releasing the interface will not delete the control itself (see `delete`), since other interfaces on that object may still be active. See the section, “Releasing Interfaces,” for more information.

Example

```
release (a)
```

send

Purpose

Returns a list of events that the control can trigger.

Syntax

```
send (a)
```

Arguments

a
Activex object returned by `actxcontrol`.

Description

Displays a list of events that controls `send`.

Example

```
send (a)
```

```
Change = Void Change ()
Click = Void Click ()
DblClick = Void DblClick ()
KeyDown = Void KeyDown (Variant(Pointer), Short)
KeyPress = Void KeyPress (Variant(Pointer), Short)
KeyUp = Void KeyUp (Variant(Pointer), Short)
MouseDown = Void MouseDown (Short, Short, Vendor-Defined,
            Vendor-Defined)
MouseMove = Void MouseMove (Short, Short, Vendor-Defined,
            Vendor-Defined)
MouseUp = Void MouseUp (Short, Short, Vendor-Defined,
            Vendor-Defined)
```

delete

Purpose

Delete an ActiveX control or server.

Syntax

```
delete (a)
```

Arguments

a

An activex object previously returned from `actxcontrol`, `actxserver`, `get`, or `invoke`.

Description

Delete an ActiveX control or server. This is different than releasing an interface, which releases and invalidates only that interface. `delete` releases all outstanding interfaces and deletes the activex server or control itself.

Example

```
delete (a)
```

Writing Event Handlers

ActiveX events are invoked when a control wants to notify its container that something of interest has occurred. For example, many controls trigger an event when the user single-clicks on the control. In MATLAB, when a control is created, you may optionally supply a callback (also known as an *event handler function*) as the last argument to the `actxcontrol` command or a cell array that contains the event handlers.

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

The event handler function (`callback`) is called whenever the control triggers any event. The event handler function must be an M-function that accepts a variable number of arguments of the following form:

```
function event (varargin)
if (str2num(vararg{1}) == –600)
    disp ('Click Event Fired');
end
```

All arguments passed to this function are MATLAB strings. The first argument to the event handler is a string that represents the number of the event that caused the event handler to be called. The remaining arguments are the values passed by the control with the event. These values will vary with the particular event and control being used. The list of events that control invocations and their corresponding event numbers and parameters must be obtained from the control's documentation. In order to use events with MATLAB, you will need to find out the numerical values that the control uses for each event so that you can use these in the event handler.

The event handlers that are used in the cell array style are slightly different than those used in the callback style. The first argument in the cell array style is a reference to the object itself. The second argument is the event number, which, unlike the callback style, is *not* a string that represents the number. (Subsequent arguments vary depending on the event.)

These sample event handlers, `myclick.m`, `my2click.m`, and `mymoused.m` correspond to the `actxcontrol` example.

```
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: ')
varargin(5)
disp('The Y position is: ')
varargin(6)
```

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

For instance

```
h = actxcontrol('SELECTOR.SelectorCtrl.1', [0 0 200 200], f, ...
{'Click' 'allevents'; 'DblClick' 'allevents'; ...
'MouseDown' 'allevents'})
```

and `allevents.m` is

```
function allevents(varargin)
if (varargin{2} = –600)
            disp ('Single Click Event Fired')
elseif (varargin{2} = –601)
            disp ('Double Click Event Fired')
elseif (varargin{2} = –605)
            disp ('Mousedown Event Fired')
end
```

Note: MATLAB does not support event arguments passed by reference or return values from events.

Additional ActiveX Client Information

Releasing Interfaces

Each ActiveX object can support one or more interfaces. In MATLAB, an interface is represented by an instance of the activex class. There are three ways to get a valid interface object into the MATLAB workspace:

- Return value from `actxcontrol`/`actxserver`
- Return value from a property via `get`
- Return value from a method invocation via `invoke`

In each case, once the interface is represented by an activex object in the workspace, it must be released when you are finished using it. Failure to release interface handles will result in memory and resources being consumed. Alternatively, you can use the `delete` command on any valid interface object, and all interfaces for that object will automatically be released (and thus invalidated), and the ActiveX server or control itself will be deleted.

MATLAB will automatically release all interfaces for an ActiveX control when the figure window that contains that control is deleted or closed. MATLAB will also automatically release all handles for an ActiveX automation server when MATLAB is shut down.

Using ActiveX Collections

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

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

This example iterates through the members of a collection. Each member of the collection is itself an interface (called `Plot` and represented by a MATLAB activex object called `hPlot`.) In particular, this example iterates through a

collection of `Plot` interfaces, invokes the `Redraw` method for each interface, and then releases each interface.

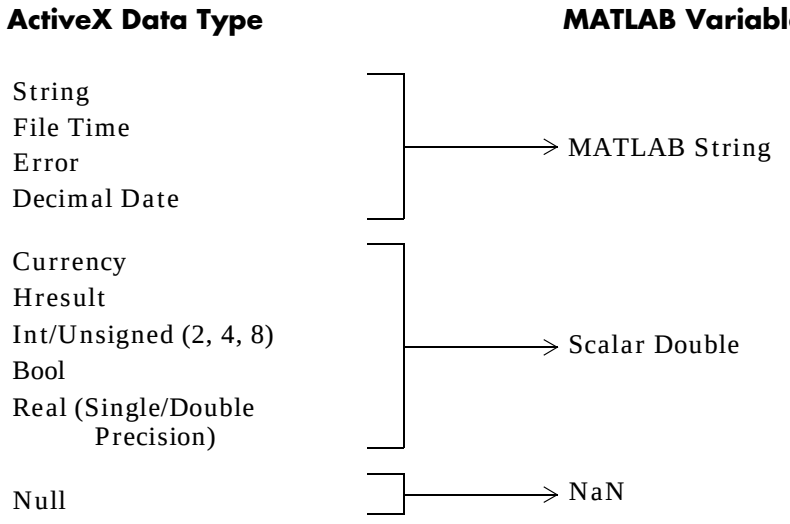
```
hCollection = get (hControl, 'Plots');
for i=1:get (hCollection, 'Count')
hPlot  = invoke (hCollection, 'Item', i);
          invoke (hPlot, 'Redraw');
          release (hPlot);
end;
release (hCollection);
```

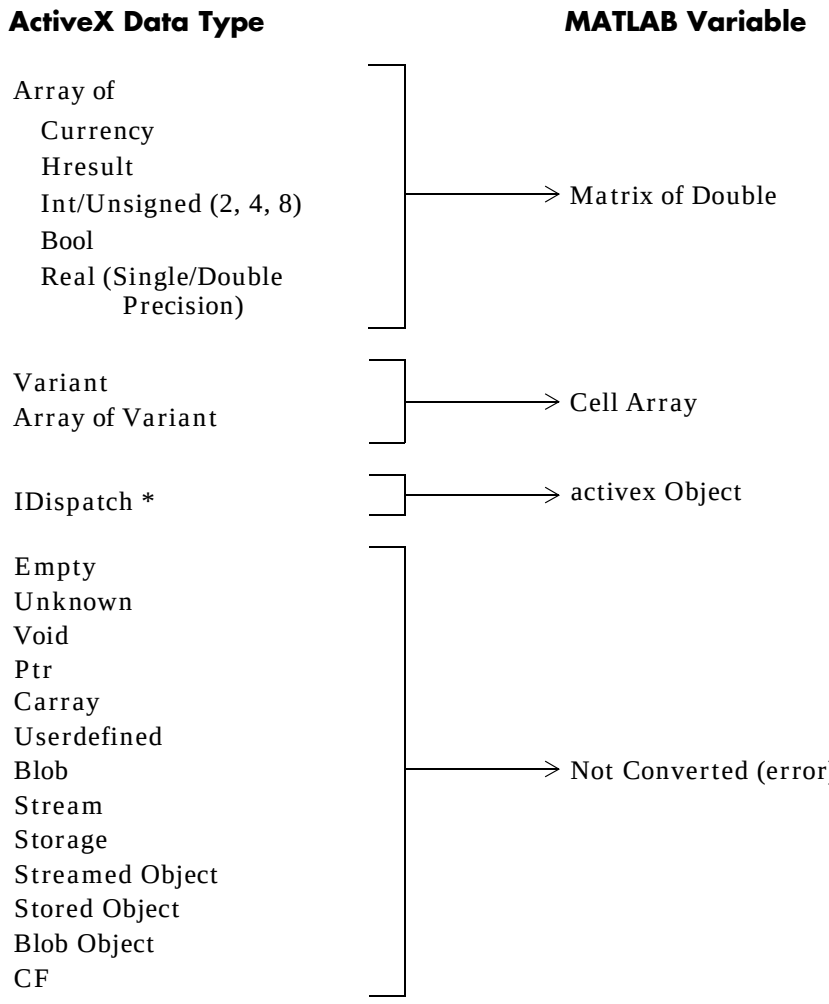
Data Conversions

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

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

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





Using MATLAB As a DCOM Server Client

Distributed Component Object Model (DCOM) is an object distribution mechanism that allows ActiveX clients to use remote ActiveX objects over a network. At the time of this writing, DCOM is shipped with NT 4.0, and can be obtained from Microsoft for Windows 95.

MATLAB has been tested as a DCOM server with Windows NT 4.0 only. 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 will appear on the remote machine.

MATLAB ActiveX Control Container Limitations

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

- MATLAB only supports indexed collections.
- ActiveX controls are not printed with figure windows.
- MATLAB does not support event arguments passed by reference.
- MATLAB does not support returning values from event handler functions.
- The position vector of a control cannot be changed or queried.
- MATLAB does not support method or event arguments passed by reference.

MATLAB Sample Control

MATLAB ships with a very simple example ActiveX control that draws a circle on the screen, displays some text, and fires a “clicked” event when the user clicks on the control. This control makes it easy to try out ActiveX control support with a known control. The control can be created by running the `mwsamp` file in the `winfun` directory.

The control is stored in the MATLAB `bin` (executable) directory along with the control’s “type library” (a binary file used by ActiveX tools to decipher the control’s capabilities).

MATLAB ActiveX Automation Server Support

MATLAB on Microsoft Windows supports ActiveX Automation server capabilities. Automation is an ActiveX protocol that allows one application or component (the “controller”) to control another application or component (the “server”). Thus, MATLAB can be launched and controlled by any Windows program that can be an Automation Controller. Some examples of applications

that can be Automation Controllers are Microsoft Excel, Microsoft Access, Microsoft Project, and many Visual Basic and Visual C++ programs. Using Automation, you can execute MATLAB commands, and get and put `mxArrays` from and to the MATLAB workspace.

To use MATLAB as an automation server, follow steps 1 and 2.

1. Consult the documentation of your controller to find out how to invoke an ActiveX Automation server. The name of the MATLAB ActiveX object that is placed in the registry is `Matlab.Application`. Exactly how you invoke the MATLAB server depends on which controller you choose, but all controllers require this name to identify the server.

2. The ActiveX Automation interface to MATLAB supports several "methods," which are described below. Here is a Visual Basic code fragment that invokes the MATLAB Automation `Execute` method, and that works in Microsoft Excel or any other Visual Basic or Visual Basic for Applications (VBA)-enabled application. The `Execute` method takes a command string as an argument and returns the results as a string. The command string can be any command that would normally be typed in the command window; the result contains any output that would have been printed to the command window as a result of executing the string, including errors.

```
Dim MatLab As Object
Dim Result As String

Set MatLab = CreateObject("Matlab.Application")
Result = MatLab.Execute("surf(peaks)")
```

MATLAB ActiveX Automation Methods

This section lists the methods that are supported by the MATLAB Automation Server. The data types for the arguments and return values are expressed as ActiveX Automation data types, which are language-independent types defined by the ActiveX Automation protocol. For example, `BSTR` is a wide-character string type defined as an Automation type, and is the same data format used by Visual Basic to store strings. Any ActiveX-compliant controller should support these data types, although the details of how you declare and manipulate these are controller specific.

```
BSTR Execute([in] BSTR Command);
```

This command accepts a single string (Command), which contains any command that can be typed at the MATLAB command window prompt. MATLAB will execute the command and return the results as a string. Any figure windows generated by the command are displayed on the screen as if the command were executed directly from the command window or an M-file. A Visual Basic example is

```
Dim MatLab As Object
Dim Result As String

Set MatLab = CreateObject("Matlab.Application")
Result = MatLab.Execute("surf(peaks)")
```

```
void GetFullMatrix(
          [in] BSTR Name,
          [in] BSTR Workspace,
          [in, out] SAFEARRAY(double)* pr,
          [in, out] SAFEARRAY(double)* pi);
```

This method retrieves a full, one- or two-dimensional real or imaginary mxArray from the named workspace. The real and (optional) imaginary parts are retrieved into separate arrays of doubles.

Name. Identifies the name of the mxArray to be retrieved.

Workspace. Identifies the workspace that contains the mxArray. Use the workspace name “base” to retrieve an mxArray from the default MATLAB workspace. Use the workspace name “global” to put the mxArray into the global MATLAB workspace. The “caller” workspace does not have any context in the API when used outside of MEX-files.

pr. Array of reals that is dimensioned to be the same size as the mxArray being retrieved. On return, this array will contain the real values of the mxArray.

pi. Array of reals that is dimensioned to be the same size as the mxArray being retrieved. On return, this array will contain the imaginary values of the mxArray. If the requested mxArray is not complex, an empty array must be

passed. In Visual Basic, an empty array is declared as `Dim Mempty() as Double`. A Visual Basic example of this method is

```
Dim MatLab As Object
Dim Result As String
Dim MReal(1, 3) As Double
Dim MImag() As Double
Dim RealValue As Double
Dim i, j As Integer

rem We assume that the connection to MATLAB exists.
Result = MatLab.Execute("a = [1 2 3 4; 5 6 7 8;]")
Call MatLab.GetFullMatrix("a", "base", MReal, MImag)

For i = 0 To 1
   For j = 0 To 3
     RealValue = MReal(i, j)
   Next j
Next i
```

```
void PutFullMatrix(
          [in] BSTR Name,
          [in] BSTR Workspace,
          [in] SAFEARRAY(double) pr,
          [in] SAFEARRAY(double) pi);
```

This method puts a full, one- or two-dimensional real or imaginary `mxArray` into the named workspace. The real and (optional) imaginary parts are passed in through separate arrays of doubles.

Name. Identifies the name of the `mxArray` to be placed.

Workspace. Identifies the workspace into which the `mxArray` should be placed. Use the workspace name “base” to put the `mxArray` into the default MATLAB workspace. Use the workspace name “global” to put the `mxArray` into the global MATLAB workspace. The “caller” workspace does not have any context in the API when used outside of MEX-files.

pr. Array of reals that contains the real values for the `mxArray`.

pi. Array of reals that contains the imaginary values for the `mxArray`. If the `mxArray` that is being sent is not complex, an empty array must be passed for

this parameter. In Visual Basic, an empty array is declared as `Dim Mempty() as Double`. A Visual Basic example of this method is

```
Dim MatLab As Object
Dim MReal(1, 3) As Double
Dim MImag() As Double
Dim i, j As Integer
For i = 0 To 1
    For j = 0 To 3
      MReal(i, j)  = I * j;
    Next j
Next I
rem We assume that the connection to MATLAB exists.
Call MatLab.PutFullMatrix("a", "base", MReal, MImag)
```

Additional ActiveX Server Information

Launching the MATLAB ActiveX Server

For MATLAB to act as an automation server, it must be started with the `/Automation` command line argument. Microsoft Windows does this automatically when an ActiveX connection is established by a controller. However, if MATLAB is already running and was launched *without* this parameter, any request by an automation controller to connect to MATLAB as a server will cause Windows to launch another instance of MATLAB with the `/Automation` parameter. This protects controllers from interfering with any interactive MATLAB sessions that may be running.

Specifying a Shared or Dedicated Server

You can launch the MATLAB Automation server in one of two modes — shared or dedicated. A dedicated server is dedicated to a single client; a shared server is shared by multiple clients.

The mode is determined by the Program ID (ProgID) used by the client to launch MATLAB. The ProgID, `Matlab.Application`, (or the version-specific ProgID, `Matlab.Application.5`) specifies the default mode, which is shared. To specify a dedicated server, use the ProgID, `Matlab.Application.Single`, (or the version-specific ProgID, `Matlab.Application.Single.5`). The term `Single` refers to the number of clients that may connect to this server.

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

Note: Clients will not connect to an interactive instance of MATLAB, that is, an instance of MATLAB that was launched manually and without the `/Automation` command line flag.

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

Using MATLAB As a DCOM Server

DCOM is a protocol that allows ActiveX connections to be established over a network. If you are using a version of Windows that supports DCOM (Windows NT 4.0 at the time of this writing) and a controller that supports DCOM, you can use the controller to launch MATLAB 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 will 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.

Dynamic Data Exchange (DDE)

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

This section describes these new DDE functions in the following order:

- DDE Concepts and Terminology
- Accessing MATLAB As a Server
- Using MATLAB As a Client
- DDE Advisory Links

DDE Concepts and Terminology

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

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

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

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

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

The Service Name

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

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

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

The Topic

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

The Item

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

Clipboard Formats

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

- *Text* – Data in Text format is a buffer of characters terminated by the `null` character. Lines of text in the buffer are delimited by a carriage return line-feed combination. If the buffer contains columns of data, those columns are delimited by the tab character. MATLAB supports Text format for

obtaining the results of a remote `EvalString` command and requests for matrix data. Also, matrix data can be sent to MATLAB in Text format.

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

Accessing MATLAB As a Server

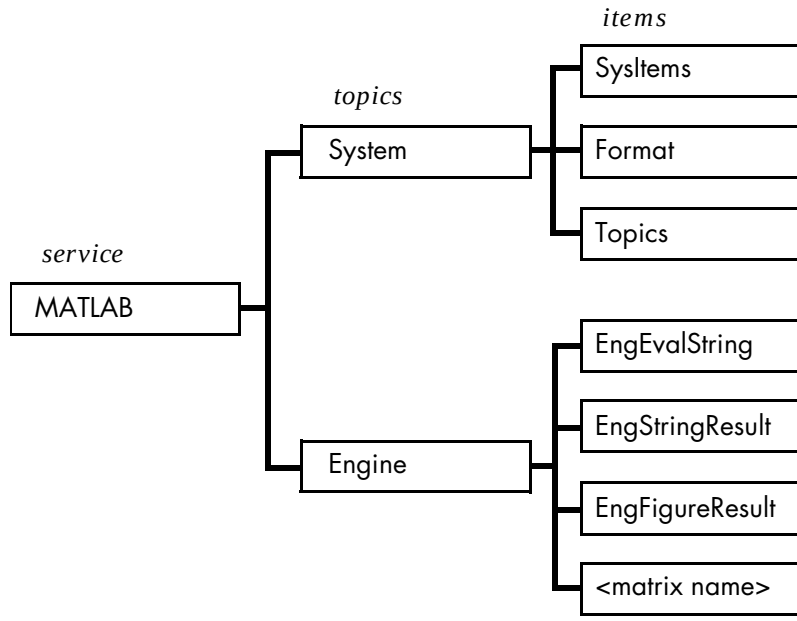
A client application can access MATLAB as a DDE server in the following ways, depending on the client application:

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

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

The DDE Name Hierarchy

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



MATLAB DDE Topics

MATLAB topics are *System* and *Engine*:

- The *System* topic allows users to browse the list of topics provided by the server, the list of *System* topic items provided by the server, and the formats supported by the server. These items are described in more detail below.
- The *Engine* topic allows users to use MATLAB as a server by passing it a command to execute, requesting data, or sending data. These items are also described in more detail below.

MATLAB System Topic Support. The MATLAB *System* topic supports these items:

- *SysItems* provides a tab-delimited list of items supported under the *System* topic (this list).
- *Format* provides a tab-delimited list of string names of all the formats supported by the server. MATLAB supports Text, Metafilepict, and XLTable. These formats are described above in the "Clipboard Formats" section.
- *Topics* provides a tab-delimited list of the names of the topics supported by MATLAB.

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

Sending Commands to MATLAB for Evaluation – Clients send commands to MATLAB using the DDE execute operation. The *Engine* topic supports DDE execute in two forms because some clients require that you specify the item name and the command to execute, while others require only the command. Where an item name is required, use `EngEvalString`. In both forms, the format of the command must be Text. Most clients default to Text for DDE execute. If the format cannot be specified, it is probably Text. The table shows a summary of the DDE execute parameters.

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

Requesting Data from MATLAB – Clients request data from MATLAB using the DDE request operation. The *Engine* topic supports DDE requests for three functions: text that is the result of the previous DDE execute command, graphical results of the previous DDE execute command, and the data for a specified matrix.

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

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

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

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

The table shows a summary of the DDE request parameters.

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

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

The table shows a summary of the DDE poke parameters.

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

Example: Using Visual Basic and the MATLAB DDE Server

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

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

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

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

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

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

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

End If
End Sub
```

Using MATLAB As a Client

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

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

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

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

If the server application is Microsoft Excel, you can specify the *System* topic or a topic that is a filename. If you specify the latter, the filename ends in `.XLS` or `.XLC` and includes the full path if necessary. A Microsoft Excel item is a cell reference, which can be an individual cell or a range of cells.

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

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

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

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

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

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

DDE Advisory Links

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

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

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

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

This example establishes a DDE conversation between MATLAB, acting as a client, and Microsoft Excel. The example extends the example in the previous section by creating a hot link with Excel. The link updates matrix `z` and evaluates a callback when the range of cells changes. A push-button, user interface control terminates the advisory link and the DDE conversation when

pressed. (For more information about creating a graphical user interface, see the MATLAB manual *Building GUIs with MATLAB*.)

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

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

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

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

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

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

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

8

System Setup

Custom Building MEX-Files

This section discusses in detail the process that the MEX-file build script uses. In general, the defaults that come with MATLAB should be sufficient for building most MEX-files. There are reasons that you might need more detailed information, such as:

- 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) 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).

The `mex` script has a set of switches that you can use to modify the link and compile stages. Table 8-1 lists the available switches and their uses.

Table 8-1: MEX Script Switches

Switch	Function
`-argcheck`	Perform argument checking on MATLAB API functions (C functions only).
`-c`	Compile only; do not link.
`-D<name>[=<def>]`	(UNIX) Define C preprocessor macro `<name>` [as having value `<def>`].
`-D<name>`	(Windows) Define C preprocessor macro `<name>`.
`-f <file>`	(UNIX and Windows) Use `<file>` as the options file; `<file>` is a full pathname if it is not in current directory. (On Windows, not necessary if you use the `-setup` option.)

Table 8-1: MEX Script Switches (Continued)

Switch	Function
–F <file>	(UNIX) Use <file> as the options file. <file> is searched for in the following order: The file that occurs first in this list is used: • ./<filename> • $HOME/matlab/<filename> • $TMW_ROOT/bin/<filename>
–F <file>	(Windows) Use <file> as the options file. (Not necessary if you use the –setup option.) <file> is searched for in the current directory first and then in the same directory as mex.bat.
–g	Build an executable with debugging symbols included.
–h[elp]	Help; lists the switches and their functions.
–I<pathname>	Include <pathname> in the compiler include search path.
–l<file>	(UNIX) Link against library lib<file>.
–L<pathname>	(UNIX) Include <pathname> in the list of directories to search for libraries.
<name>=<def>	(UNIX) Override options file setting for variable <name>.
–n	No execute flag. Using this option causes the commands that would be used to compile and link the target to be displayed without executing them.
–output <name>	Create an executable named <name>. (An appropriate executable extension is automatically appended.)
–O	Build an optimized executable.

Table 8-1: MEX Script Switches (Continued)

Switch	Function
–setup	(Windows) Set up default options file. This switch should be the only argument passed.
–U<name>	(UNIX and Windows) Undefine C preprocessor macro <name>.
–V4	Compile MATLAB 4-compatible MEX-files.
–v	Verbose; print all compiler and linker settings.

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.

Locating the Default Options File

For UNIX, the default options file provided with MATLAB is located in <matlab>/bin. For Windows, the default options file is in <matlab>\bin.

On UNIX, the mex script will look for an options file called mexopts.sh in the current directory first. It searches next in your $HOME/matlab directory, and finally in <matlab>/bin. On Windows, the mex script assumes that the options file, mexopts.bat, is in the same directory as mex.bat, i.e., <matlab>\bin. On both platforms, you can directly specify the name of the options file using the –f switch.

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file in <matlab>/bin/mexopts.sh (<matlab>\bin\mexopts.bat in Windows), or you can invoke the mex script in verbose mode.

The following section provides additional details regarding each of these stages. However, there is a general way to obtain specifics on the build process, which is the verbose option to the mex script (the –v flag). This will print the exact compiler options, prelink commands (if appropriate), and linker options used. The following section gives an overview of the high-level process; for

exact flags provided for each compiler, invoke the `mex` script with the verbose flag.

UNIX

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

- Add `<matlab>/extern/include` to the list of directories in which to find header files (`–I<matlab>/extern/include`)
- Define the preprocessor macro `MATLAB_MEX_FILE` (`–DMATLAB_MEX_FILE`)
- (C MEX-files only) Compile the source file, which contains version information for the MEX-file, `<matlab>/extern/src/mexversion.c`

For all platforms except SunOS 4.x, the link stage must:

- Instruct the linker to build a shared library
- Link all objects from compiled source files (including `mexversion.c`)
- (Fortran MEX-files only) Link in the precompiled versioning source file, `<matlab>/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 lower case and may have appended underscores. For specific information, invoke the `mex` script in verbose mode and examine the output.

On the SunOS 4.x platform, the link stage is more complicated. The `mex` script does a test run of the linker to see what libraries need to be linked in, and what flags need to be used. The output of the test run, and the final flags and libraries used, are different for each compiler and compiler version. However, they are displayed in the verbose output.

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 `<matlab>/bin`.

The mex script looks for an options file called mexopts.sh first in the current directory, then in your $HOME/matlab directory, and finally in <matlab>/bin. You can also directly specify the name of the options file using the –f option.

To aid in providing flexibility, there are two sets of options in the options file that can be turned 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 will include the OPTIMFLAGS options in the compile and link stages.
- If you build in debug mode, the mex script will include the DEBUGFLAGS options in the compile and link stages, but will 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 can also provide explicit lists of libraries that must be linked in to all MEX-files containing source files of each language.

The variables can be summed up as 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	----

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

- Examine the options file in <matlab>/bin/mexopts.sh (or the options file you are using), or
- Invoke the mex script in verbose mode

Windows

There are three stages to MEX-file building for both C and Fortran on Windows – compiling, prelinking, and linking. The compile stage 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 remark 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 COMPLFLAGS environment variable.
  - **a** The switch to create a DLL is required for MEX-files.
  - **b** For stand-alone programs, the switch to create an exe is required.
  - **c** The –c switch (compile only; do not link) is recommended.
  - **d** The switch to specify 8-byte alignment.
  - **e** Any other switch specific to the environment can be used.

- Define preprocessor macro, with –D, MATLAB_MEX_FILE is required.
- Set up optimizer switches and/or debug switches using OPTIMFLAGS and DEBUGFLAGS. These are mutually exclusive: the OPTIMFLAGS are the default, and the DEBUGFLAGS are used if you set the –g switch on the mex command line.

The prelink stage dynamically creates import libraries to import the required function into the MEX, MAT, or engine file. All MEX-files link against MATLAB only. MAT stand-alone programs link against libmx.dll (array access library) and libmat.dll (MAT-functions). Engine stand-alone programs link against libmx.dll (array access library) and libeng.dll for engine functions. MATLAB and each DLL have corresponding .def files of the same names located in the <matlab>\extern\include directory.

Finally, the link stage 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 stand-alone programs.
  - Export of the entry point to the MEX-file as mexFunction for C or MEXFUNCTION@16 for Microsoft Fortran.
  - The import library(s) created in the PRELINK_CMDS stage.
  - Any other link switch specific to the compiler that can be used.
- Define the linking optimization switches and debugging switches in LINKEROPTIMFLAGS and LINKDEBUGFLAGS. As in the compile stage, these two are mutually exclusive: the default is optimization, and the –g switch invokes the debug switches.
- Define the link-file identifier in the LINK_FILE environment variable, if needed. 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 needed. 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, it can be overridden by specifying the mex –output switch.

Linking DLLs to MEX-Files

To link a DLL to a MEX-file, list the DLL on the command line and make sure that the PRELINK_DLLS command is properly completed in the options file. The PRELINK_DLLS command dynamically creates an import library from your DLL so that the DLL can be linked with your MEX-file. PRELINK_DLLS contains the import library creation command and options for your compiler and uses the variable DLL_NAME, which is assigned to the DLL name(s) provided on the command line.

Versioning MEX-Files

The mex 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 will be linked into the MEX-file using the standard link command.
- If a linker command is given, the resource file will be linked 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 MEX-files in the Microsoft Visual C++ (MSVC) 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 MSVC 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 and `mexversion.rc` into it.

2. Create a `.DEF` file to export the MEX entry point. For example

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

3. Add the `.DEF` file to the project.

4. Create an import library of MEX-functions from `MATLAB.DEF` using the `LIB` command. For example

```
LIB /DEF:MATLAB\EXTERN\INCLUDE\MATLAB.DEF/OUT:mymeximports.lib
```

5. Add the import library to the library modules in the `LINK` settings option.

6. Add the MATLAB `include` directory, `MATLAB\EXTERN\INCLUDE` to the `include` path in the **Settings C/C++ Preprocessor** option.

7. Add `MATLAB_MEX_FILE` to the **C/C++ Preprocessor** option by selecting **Settings** from the **Build** menu, selecting **C/C++**, and then typing `,MATLAB_MEX_FILE` after the last entry in the **Preprocessor definitions** field.

8. To debug the MEX-file using the IDE, put `MATLAB.EXE` in the **Settings Debug** option as the **Executable for debug session**.

Troubleshooting

MEX-File Creation

Use Figure 8-1 to help isolate difficulties in creating MEX-files. The following section, “Understanding MEX-File Problems,” provides additional information regarding common problems that occur when creating MEX-files. If the suggestions in these sections do not help, access the Solutions Search Engine at `http://www.mathworks.com/solution.html`.

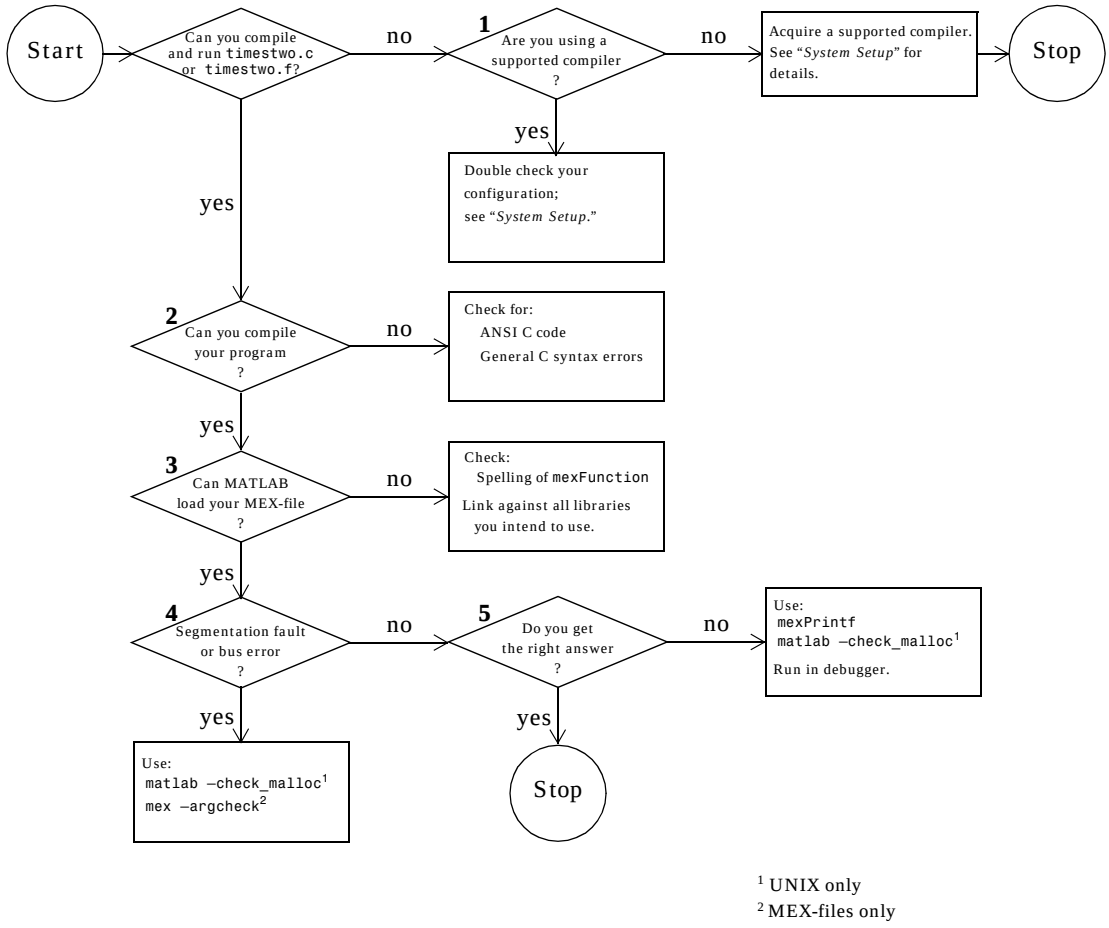


Figure 8-1: Troubleshooting MEX-File Creation Problems

Understanding MEX-File Problems

This section contains information regarding common problems that occur when creating MEX-files. Problems 1 through 5 refer to specific sections of the

previous flowchart, and the remaining sections refer to a particular platform situations.

Problem 1

The most common configuration problem in creating C MEX-files on UNIX 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 The MathWorks generate a string of syntax errors when you try to compile your code. See "Building MEX-Files" in Chapter 2 for information on selecting the appropriate options file or, if necessary, obtain an ANSI C compiler.

Problem 2

A second way of generating a string of syntax errors occurs when you attempt to mix ANSI and non-ANSI C code. The MathWorks 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.

Problem 3

If you receive an error of the form

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

MATLAB is unable to recognize your MEX-file as being valid.

MATLAB loads MEX-files by looking for the gateway routine, `mexFunction`. If you misspell the function name, MATLAB is not able to load your MEX-file and generates an error message. On Windows, 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, you see a system error message referring to

"unresolved symbols" or "unresolved references." Be sure to link against the library that defines the function in question.

On Windows, MATLAB will fail to load MEX-files if it cannot find all DLLs referenced by the MEX-file; the DLLs must be on the path or in the same directory as the MEX-file. This is also true for third party DLLs.

Problem 4

If your MEX-file causes a segmentation violation or bus error, it means that 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 in troubleshooting problems of this nature. Listed in order of simplicity, they are:

- Recompile your 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 such errors as passing `null` pointers to API functions.
- Run MATLAB with the `–check_malloc` option (UNIX only). 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 such errors 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 MEX-files, respectively.

Problem 5

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 possibility for 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.

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.

so_locations Error on SGI

When compiling a MEX-file under MATLAB 5 on SGI systems, you may get the following errors:

```
ld error 48 cannot acces registry file so_locations no locks
available - ignored.
fatal 51 - can't assign virtual addresses for filename.mexsg
within specified range.
```

The linker creates a file called `so_locations`. This file is typically deleted, however, in some cases it may not be deleted on your system. Deleting this file should resolve the problem. However, if the workaround does not solve the problem, try the following:

1. Move the file you are trying to compile to a `/tmp` directory.
2. Try compiling the file from this new location.

If the file compiles in the `/tmp` directory, there is a problem with your NFS network configuration. The problem is that `ld` (system linker) cannot place `LOCKS` files on NFS drives. If this is the case, see your system administrator and make sure that `LOCKD` is running on your NFS server. For more information, you may want to contact SGI directly.

Memory Management Compatibility Issues

To address performance issues, we have made some changes to the internal MATLAB memory management model. These changes will allow us to provide future enhancements to the MEX-file API.

As of MATLAB 5.2, MATLAB implicitly calls `mxDestroyArray`, the `mxArray` destructor, at the end of a MEX-file's execution on any `mxArrays` that are not returned in the left-hand side list (`plhs[]`). You are now warned if MATLAB detects any misconstructed or improperly destructed `mxArrays`.

We highly recommend that you fix code in your MEX-files that produces any of the warnings discussed in the following sections. For additional information, see the section, "Memory Management," in Chapter 3.

Improperly Destroying an mxArray

You cannot use `mxFree` to destroy an `mxArray`.

Warning

```
Warning:  You are attempting to call mxFree on a <class-id> array.
The destructor for mxArrays is mxDestroyArray; please call this
instead. MATLAB will attempt to fix the problem and continue, but
this will result in memory faults in future releases.
```

Note: Currently these warnings are enabled by default for backwards compatibility reasons. In future releases of MATLAB, the warnings will be disabled by default. The programmer will be responsible for enabling these warnings during the MEX-file development cycle.

Example That Causes Warning

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

`mxFree` does not destroy the array object. This operation frees the structure header associated with the array, but MATLAB will still operate as if the array object needs to be destroyed. Thus MATLAB will try to destroy the array object, and in the process, attempt to free its structure header again.

Solution

Call `mxDestroyArray` instead.

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

Incorrectly Constructing a Cell or Structure mxArray

You cannot call `mxSetCell` or `mxSetField` variants with `prhs[]` as the member array.

Warning

```
Warning: You are attempting to use an array from another scope
(most likely an input argument) as a member of a cell array or
structure. You need to make a copy of the array first. MATLAB will
attempt to fix the problem and continue, but this will result in
memory faults in future releases.
```

Example That Causes Warning

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

    mxArray *temp = mxCreateCellMatrix(1,1);
      ...
    mxSetCell(temp, 0, prhs[0]);  /* INCORRECT */
```

When the MEX-file returns, MATLAB will destroy the entire cell array. Since this includes the members of the cell, this will implicitly destroy 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).

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, 0, mxDuplicateArray(prhs[0]));  /* CORRECT */
```

Creating a Temporary mxArray with Improper Data

You cannot call `mxDestroyArray` on an `mxArray` whose data was not allocated by an API routine.

Warning

```
Warning: You have attempted to point the data of an array to a
block of memory not allocated through the MATLAB API. MATLAB will
attempt to fix the problem and continue, but this will result in
memory faults in future releases.
```

Example That Causes Warning

If you call `mxSetPr`, `mxSetPi`, `mxSetData`, or `mxSetImagData` with memory as the intended data block (second argument) that was not allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`:

```
    mxArray *temp = mxCreateDoubleMatrix(0,0,mxREAL);
```

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

then when the MEX-file returns, MATLAB will attempt to free the pointer to real data and the pointer to imaginary data (if any). Thus MATLAB will attempt to free memory, in this example, from the program stack. This will cause the above warning when MATLAB attempts to reconcile its consistency checking information.

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 */
```

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 would still free 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

```
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 address this issue as shown above to avoid such memory leaks.

MEX-Files Should Destroy Their Own Temporary Arrays

In general, we recommend that MEX-files destroy their own temporary arrays and clean up their own temporary memory. All `mxArrays` except those returned in the left-hand side list and those returned by `mexGetArrayPtr` may be safely destroyed. 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.)

A

API Functions

C MX-Functions

```
char *mxArrayToString(const mxArray *array_ptr)
void mxAssert(int expr, char *error_message)
void mxAssertS(int expr, char *error_message)
int mxCalcSingleSubscript(const mxArray *array_ptr,
            int nsubs, int *subs)
void *mxCalloc(size_t n, size_t size)
void mxClearLogical(mxArray *array_ptr)
typedef enum mxComplexity(mxREAL=0, mxCOMPLEX)
mxArray *mxCreateCellArray(int ndim, const int *dims)
mxArray *mxCreateCellMatrix(int m, int n)
mxArray *mxCreateCharArray(int ndim, const int *dims)
mxArray *mxCreateCharMatrixFromStrings(int m,
            const char **str)
mxArray *mxCreateDoubleMatrix(int m, int n,
            mxComplexity ComplexFlag)
mxCreateFull (Obsolete)
mxArray *mxCreateNumericArray(int ndim, const int *dims,
             mxClassID class, mxComplexity ComplexFlag)
mxArray *mxCreateSparse(int m, int n, int nzmax,
             mxComplexity ComplexFlag)
mxArray *mxCreateString(const char *str)
mxArray *mxCreateStructArray(int ndim, const int *dims,
             int nfields,const char **field_names)
mxArray *mxCreateStructMatrix(int m, int n, int nfields,
             const char **field_names)
void mxDestroyArray(mxArray *array_ptr)
mxArray *mxDuplicateArray(const mxArray *in)
mxArray *mxFree(void *ptr)
mxFreeMatrix (Obsolete)
mxArray *mxGetCell(const mxArray *array_ptr, int index)
mxClassID mxGetClassID(const mxArray *array_ptr)
const char *mxGetClassName(const mxArray *array_ptr)
void *mxGetData(const mxArray *array_ptr)
const int *mxGetDimensions(const mxArray *array_ptr)
int mxGetElementSize(const mxArray *array_ptr)
double mxGetEps(void)
mxArray *mxGetField(const mxArray *array_ptr, int index,
            const char *field_name)
```

```
mxArray *mxGetFieldByNumber(const mxArray *array_ptr,
            int index, int *field_number)
const char *mxGetFieldNameByNumber(const mxArray *array_ptr,
            int field_number)
int mxGetFieldNumber(const mxArray *array_ptr,
            const char *field_name)
void *mxGetImagData(const mxArray *array_ptr)
double mxGetInf(void)
int *mxGetIr(const mxArray *array_ptr)
int *mxGetJc(const mxArray *array_ptr)
int mxGetM(const mxArray *array_ptr)
int mxGetN(const mxArray *array_ptr)
const char *mxGetName(const mxArray *array_ptr)
double mxGetNaN(void)
int mxGetNumberOfDimensions(const mxArray *array_ptr)
int mxGetNumberOfElements(const mxArray *array_ptr)
int mxGetNumberOfFields(const mxArray *array_ptr)
int mxGetNzmax(const mxArray *array_ptr)
double *mxGetPi(const mxArray *array_ptr)
double *mxGetPr(const mxArray *array_ptr)
double mxGetScalar(const mxArray *array_ptr)
int mxGetString(const mxArray *array_ptr, char *buf,
            int buflen)
bool mxIsCell(const mxArray *array_ptr)
bool mxIsChar(const mxArray *array_ptr)
bool mxIsClass(const mxArray *array_ptr, const char *name)
bool mxIsComplex(const mxArray *array_ptr)
bool mxIsDouble(const mxArray *array_ptr)
bool mxIsEmpty(const mxArray *array_ptr)
bool mxIsFinite(double value)
bool mxIsFromGlobalWS(const mxArray *array_ptr)
mxIsFull (Obsolete)
bool mxIsInf(double value)
bool mxIsInt8(const mxArray *array_ptr)
bool mxIsInt16(const mxArray *array_ptr)
bool mxIsInt32(const mxArray *array_ptr)
bool mxIsLogical(const mxArray *array_ptr)
bool mxIsNaN(double value)
bool mxIsNumeric(const mxArray *array_ptr)
bool mxIsSingle(const mxArray *array_ptr)
```

```
bool mxIsSparse(const mxArray *array_ptr)
mxIsString (Obsolete)
bool mxIsStruct(const mxArray *array_ptr)
bool mxIsUint8(const mxArray *array_ptr)
bool mxIsUint16(const mxArray *array_ptr)
bool mxIsUint32(const mxArray *array_ptr)
void *mxMalloc(size_t n)
void *mxRealloc(void *ptr, size_t size)
void mxSetAllocFcns(calloc_proc callocfcn, free_proc freefcn,
             realloc_proc reallocfcn, malloc_proc mallocfcn)
void mxSetCell(mxArray *array_ptr, int index, mxArray *value)
int mxSetClassName(mxArray *array_ptr, const char *classname)
void mxSetData(mxArray *array_ptr, void *data_ptr)
int mxSetDimensions(mxArray *array_ptr, const int *dims,
             int ndims)
void mxSetField(mxArray *array_ptr, int index,
             const char *field_name, mxArray *value)
void mxSetFieldByNumber(mxArray *array_ptr, int index,
             int field_number, mxArray *value)
void mxSetImagData(mxArray *array_ptr, void *pi)
void mxSetIr(mxArray *array_ptr, int *ir)
void mxSetJc(mxArray *array_ptr, int *jc)
void mxSetLogical(mxArray *array_ptr)
void mxSetM(mxArray *array_ptr, int m)
void mxSetN(mxArray *array_ptr, int n)
void mxSetName(mxArray *array_ptr, const char *name)
void mxSetNzmax(mxArray *array_ptr, int nzmax)
void mxSetPi(mxArray *array_ptr, double *pi)
void mxSetPr(mxArray *array_ptr, double *pr)
```

C MEX-Functions

```
void mexAddFlops(int count)
int mexAtExit(void (*ExitFcn)(void))
int mexCallMATLAB(int nlhs, mxArray *plhs[], int nrhs,
             mxArray *prhs[], const char *command_name)
void mexErrMsgTxt(const char *error_msg)
int mexEvalString(const char *command)
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
             const mxArray *prhs[])
```

```
const char *mexFunctionName(void)
const mxArray *mexGet(double handle, const char *property)
mxArray *mexGetArray(const char *name, const char *workspace)
const mxArray *mexGetArrayPtr(const char *name,
              const char *workspace)
mexGetEps (Obsolete)
mexGetFull (Obsolete)
mexGetGlobal (Obsolete)
mexGetInf (Obsolete)
mexGetMatrix (Obsolete)
mexGetMatrixPtr (Obsolete)
mexGetNaN (Obsolete)
mexIsFinite (Obsolete)
bool mexIsGlobal(const mxArray *array_ptr)
mexIsInf (Obsolete)
bool mexIsLocked(void)
mexIsNaN (Obsolete)
void mexLock(void)
void mexMakeArrayPersistent(mxArray *array_ptr)
void mexMakeMemoryPersistent(void *ptr)
int mexPrintf(const char *format, ...)
int mexPutArray(mxArray *array_ptr, const char *workspace)
mexPutFull (Obsolete)
mexPutMatrix (Obsolete)
int mexSet(double handle, const char *property,
              mxArray *value)
void mexSetTrapFlag(int trap_flag)
void mexUnlock(void)
void mexWarnMsgTxt(const char *warning_msg)
```

C MAT-File Routines

```
int matClose(MATFile *mfp)
int matDeleteArray(MATFile *mfp, const char *name)
matDeleteMatrix (obsolete)
mxArray *matGetArray(MATFile *mfp, const char *name)
mxArray *matGetArrayHeader(MATFile *mfp, const char *name)
char **matGetDir(MATFile *mfp, int *num)
FILE *matGetFp(MATFile *mfp)
matGetFull (obsolete)
```

matGetMatrix (obsolete)
mxArray ***matGetNextArray**(MATFile *mfp)
mxArray ***matGetNextArrayHeader**(MATFile *mfp)
matGetNextMatrix (*Obsolete*)
matGetString (*Obsolete*)
MATFile ***matOpen**(const char *filename, const char *mode)
int **matPutArray**(MATFile *mfp, const mxArray *mp)
int **matPutArrayAsGlobal**(MATFile *mfp, const mxArray *mp)
matPutFull (*Obsolete*)
matPutMatrix (*Obsolete*)
matPutString (*Obsolete*)

C Engine Routines

int **engClose**(Engine *ep)
int **engEvalString**(Engine *ep, const char *string)
mxArray ***engGetArray**(Engine *ep, const char *name)
engGetFull (obsolete)
engGetMatrix (obsolete)
Engine ***engOpen**(const char *startcmd)
int **engOutputBuffer**(Engine *ep, char *p, int n)
int **engPutArray**(Engine *ep, const mxArray *mp)
engPutFull (obsolete)
engPutMatrix (obsolete)
engSetEvalCallback (obsolete)
engSetEvalTimeout (obsolete)
engWinInit (obsolete)

Fortran MX-Functions

integer*4 function **mxCalloc**(n, size)
subroutine **mxCopyCharacterToPtr**(y, px, n)
subroutine **mxCopyComplex16ToPtr**(y, pr, pi, n)
subroutine **mxCopyInteger4ToPtr**(y, px, n)
subroutine **mxCopyPtrToCharacter**(px, y, n)
subroutine **mxCopyPtrToComplex16**(pr, pi, y, n)
subroutine **mxCopyPtrToInteger4**(px, y, n)
subroutine **mxCopyPtrToPtrArray**(px, y, n)
subroutine **mxCopyPtrToReal8**(px, y, n)
subroutine **mxCopyReal8ToPtr**(y, px, n)

```
integer*4 function mxCreateFull(m, n, ComplexFlag)
integer*4 function mxCreateSparse(m, n, nzmax, ComplexFlag)
integer*4 function mxCreateString(str)
subroutine mxFree(ptr)
subroutine mxFreeMatrix(pm)
integer*4 function mxGetIr(pm)
integer*4 function mxGetJc(pm)
integer*4 function mxGetM(pm)
integer*4 function mxGetN(pm)
character*32 function mxGetName(pm)
integer*4 function mxGetNzmax(pm)
integer*4 function mxGetPi(pm)
integer*4 function mxGetPr(pm)
real*8 function mxGetScalar(pm)
integer*4 function mxGetString(pm, str, strlen)
integer*4 function mxIsComplex(pm)
integer*4 function mxIsDouble(pm)
integer*4 function mxIsFull(pm)
integer*4 function mxIsNumeric(pm)
integer*4 function mxIsSparse(pm)
integer*4 function mxIsString(pm)
subroutine mxSetIr(pm, ir)
subroutine mxSetJc(pm, jc)
subroutine mxSetM(pm, m)
subroutine mxSetN(pm, n)
subroutine mxSetName(pm, name)
subroutine mxSetNzmax(pm, nzmax)
subroutine mxSetPi(pm, pi)
subroutine mxSetPr(pm, pr)
```

Fortran MEX-Functions

```
integer*4 function mexAtExit(ExitFcn)
integer*4 function mexCallMATLAB(nlhs, plhs, nrhs, prhs, name)
subroutine mexErrMsgTxt(error_msg)
integer*4 function mexEvalString(command)
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
real*8 function mexGetEps()
integer*4 function mexGetFull(name, m, n, pr, pi)
integer*4 function mexGetGlobal(name)
```

```
real*8 function mexGetInf()
integer*4 function mexGetMatrix(name)
integer*4 function mexGetMatrixPtr(name)
real*8 function mexGetNaN()
integer*4 function mexIsFinite(value)
integer*4 function mexIsInf(value)
integer*4 function mexIsNaN(value)
subroutine mexPrintf(message)
integer*4 function mexPutFull(name, m, n, pr, pi)
integer*4 function mexPutMatrix(mp)
subroutine mexSetTrapFlag(trap_flag)
```

Fortran MAT-File Routines

```
integer*4 function matClose(mfp)
subroutine matDeleteMatrix(mfp, name)
integer*4 function matGetDir(mfp, num)
integer*4 function matGetFull(mfp, name, m, n, pr, pi)
integer*4 function matGetMatrix(mfp, name)
integer*4 function matGetNextMatrix(mfp)
integer*4 function matGetString(mfp, name, str, strlen)
integer*4 function matOpen(filename, mode)
integer*4 function matPutFull(mfp, name, m, n, pr, pi)
integer*4 function matPutMatrix(mfp, mp)
integer*4 function matPutString(mfp, name, str)
```

Fortran Engine Routines

```
integer*4 function engClose(ep)
integer*4 function engEvalString(ep, command)
integer*4 function engGetFull(ep, name, m, n, pr, pi)
integer*4 function engGetMatrix(ep, name)
integer*4 function engOpen(startcmd)
integer*4 function engOutputBuffer(ep, p)
integer*4 function engPutFull(ep, name, m, n, pr, pi)
integer*4 function engPutMatrix(ep, mp)
```

DDE Routines

```
rc = ddeadv(channel, item, callback, upmtx, format, timeout)
rc = ddeexec(channel, command, item, timeout)
channel = ddeinit(service, topic)
rc = ddepoke(channel, item, data, format, timeout)
data = ddereq(channel, item, format, timeout)
rc = ddeterm(channel)
rc = ddeunadv(channel, item, format, timeout)
```

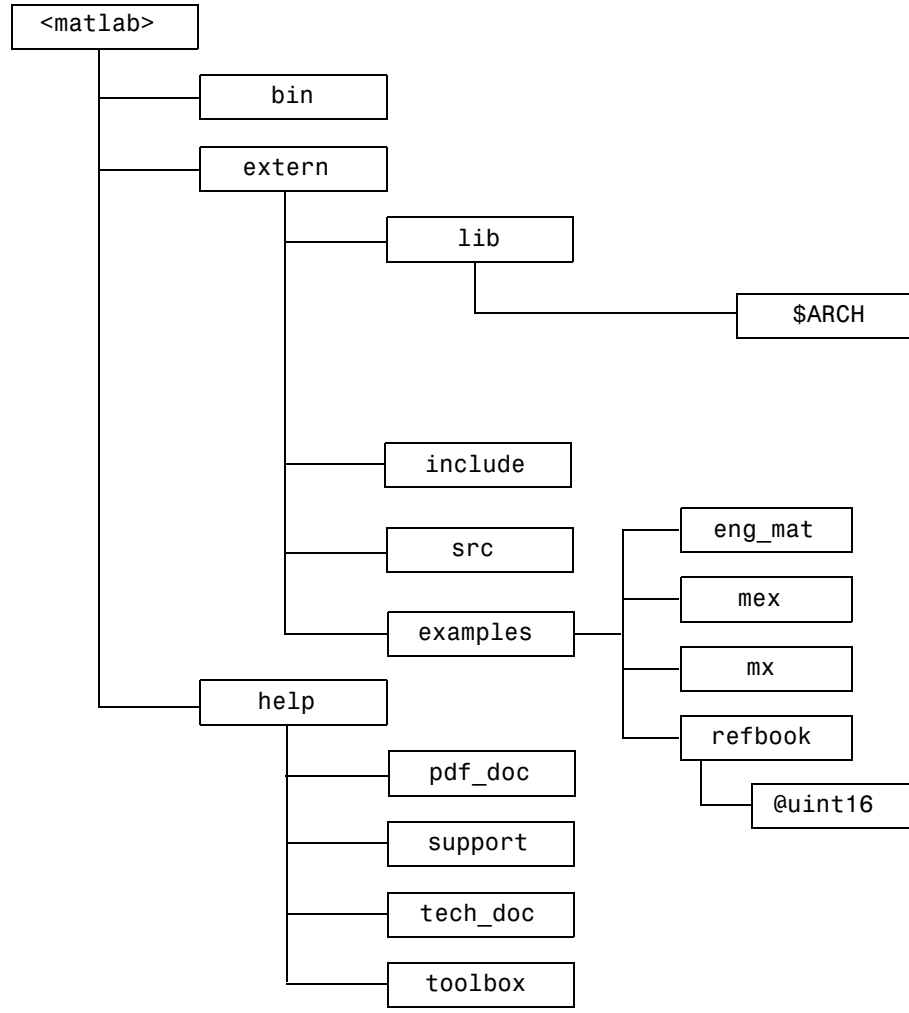
B

Directory Organization

This appendix describes the directory organization and purpose of the files associated with the MATLAB API on UNIX and Microsoft Windows systems.

Directory Organization on UNIX

This figure illustrates the directories in which the MATLAB API files are located.



In the illustration, `<matlab>` symbolizes the top-level directory where MATLAB is installed on your system.

<matlab>/bin

The <matlab>/bin directory contains two files that are relevant for the MATLAB API:

mex	UNIX shell script that creates MEX-files from C or Fortran MEX-file source code. See the *Application Program Interface Guide* for more details on mex.
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. Table B-1 lists the options files.

Table B-1: Preconfigured Options Files

mexopts.sh	System ANSI Compiler
gccopts.sh	GCC (GNU C Compiler)
cxxopts.sh	System C++ Compiler

<matlab>/extern

<matlab>/extern/lib/$ARCH

The <matlab>/extern/lib/*$ARCH* directory contains libraries, where *$ARCH* specifies a particular UNIX platform. For example, on a Sun SPARCstation running SunOs 4, the $ARCH directory is named sun4.

On some UNIX platforms, this directory contains two versions of this library. Library filenames ending with .a are static libraries and filenames ending with .so or .sl are shared libraries.

<matlab>/extern/include

The <matlab>/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.

<matlab>/extern/src

The `<matlab>/extern/src` directory contains those C source files that are necessary to support certain MEX-file features such as argument checking and versioning.

<matlab>/extern/examples/eng_mat

The `<matlab>/extern/examples/eng_mat` directory contains examples for using the MATLAB engine facility, as well as examples for reading and writing MATLAB data files (MAT-files). These examples are all stand-alone programs.

<matlab>/extern/examples/mex

The `<matlab>/extern/examples/mex` directory contains MEX-file examples. It includes the examples described in the online API reference pages for MEX interface functions (the functions beginning with the `mex` prefix).

<matlab>/extern/examples/mx

The `<matlab>/extern/examples/mx` directory contains examples for using the array access functions. Although you can use these functions in stand-alone programs, most of these are MEX-file examples. The exception is `mxSetAllocFcns.c`, since this function is available only to stand-alone programs.

<matlab>/extern/examples/refbook

The `<matlab>/extern/examples/refbook` directory contains the examples that are discussed in the *Application Program Interface Guide*.

<matlab>/extern/examples/refbook/@uint16

The <matlab>/extern/examples/refbook/@uint16 directory contains the EQ and NEQ overloaded functions.

Table B-2: Overloaded Functions

EQ	EQ overloaded function “equal” for uint16 type
NEQ	NEQ overloaded function “not equal” for uint16 type

<matlab>/help

<matlab>/help/pdf_doc

The <matlab>/help/pdf_doc directory contains online help files for MATLAB and other toolboxes in PDF format.

<matlab>/help/support

The <matlab>/help/support directory contains a World Wide Web link to the MathWorks Technical Support Department’s troubleshooting tools such as:

- Solution Search Engine
- FAQ (Frequently Asked Questions)
- Technical Notes

<matlab>/help/tech_doc

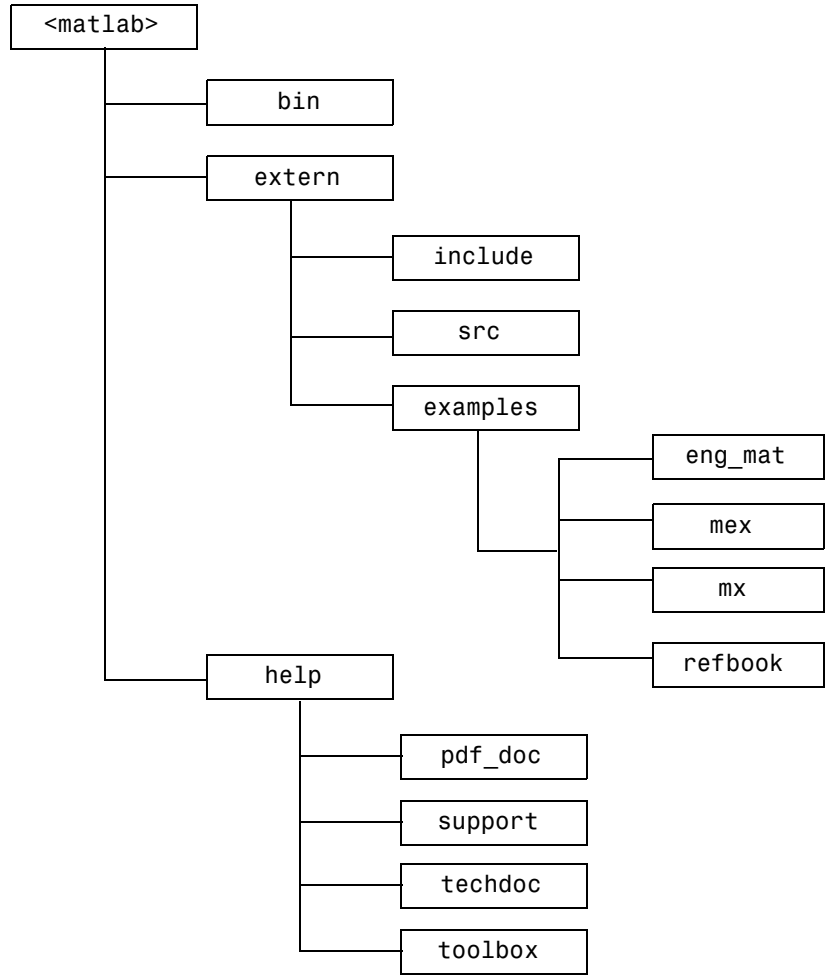
The <matlab>/help/techdoc directory contains additional technical documentation for MATLAB in HTML format, viewable through the MATLAB Help Desk.

<matlab>/help/toolbox

The <matlab>/help/toolbox directory contains online help documentation for MATLAB and other toolboxes in HTML format, viewable through the MATLAB Help Desk.

Directory Organization on Windows

This figure illustrates the directories in which the MATLAB API files are located.



In the illustration, `<matlab>` symbolizes the top-level directory where MATLAB is installed on your system.

<matlab>\bin

The <matlab>\bin directory contains the mex.bat batch file that builds C and Fortran files into MEX-files. This directory also contains the preconfigured options files that the mex script uses with particular compilers. See Table 2-2 in Chapter 2 for a complete list of the options files.

<matlab>\extern

<matlab>\extern\include

The <matlab>\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 Borland compiler.
*.def	Files used by MSVC and Microsoft Fortran compilers.
mexversion.rc	Resource file for inserting versioning information into MEX-files.

<matlab>\extern\src

The <matlab>\extern\src directory contains files that are used for debugging MEX-files.

<matlab>\extern\examples\eng_mat

The <matlab>\extern\examples\eng_mat directory contains examples for using the MATLAB engine facility, as well as examples for reading and writing MATLAB data files (MAT-files). These examples are all stand-alone programs.

<matlab>\extern\examples\mex

The <matlab>\extern\examples\mex directory contains MEX-file examples. It includes the examples described in the online API reference pages for MEX interface functions (the functions beginning with the mex prefix).

<matlab>\extern\examples\mx

The <matlab>\extern\examples\mx directory contains examples for using the array access functions. Although you can use these functions in stand-alone programs, most of these are MEX-file examples. The exception is mxSetAllocFcns.c, since this function is available only to stand-alone programs.

<matlab>\extern\examples\refbook

The <matlab>\extern\examples\refbook directory contains the examples that are discussed in the *Application Program Interface Guide*.

<matlab>\help

<matlab>\help\pdf_doc

The <matlab>\help\pdf_doc directory contains online help files for MATLAB and other toolboxes in PDF format.

<matlab>\help\support

The <matlab>\help\support directory contains a World Wide Web link to the MathWorks Technical Support Department's troubleshooting tools such as:

- Solution Search Engine
- FAQ (Frequently Asked Questions)
- Technical Notes

<matlab>\help\techdoc

The <matlab>\help\techdoc directory contains additional technical documentation for MATLAB in HTML format, viewable through the MATLAB Help Desk.

<matlab>\help\toolbox

The `<matlab>\help\toolbox` directory contains online help documentation for MATLAB and other toolboxes in HTML format, viewable through the MATLAB Help Desk.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Y