Calling C and Fortran Programs from MATLAB

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 interface to external programs written in the C and Fortran languages.

Introducing 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.

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. This table lists the platform-specific extensions for MEX-files.

Table 3-1: MEX-File Extensions

Platform	MEX-File Extension
HP-UX	`mexhpux`
Linux	`mexglx`
Macintosh	`mexmac`
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 the MATLAB 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.

MATLAB Data

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

- “The MATLAB Array”
- “Data Storage”
- “Data Types in MATLAB”
- “Using Data Types”

The MATLAB Array

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

- Its type
- Its dimensions
- The data associated with this array
- If numeric, whether the variable is real or complex
- If sparse, its indices and nonzero maximum elements
- If a structure or object, the number of fields and field names

Data Storage

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

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

its dimensions are

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

and its data is stored as

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

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.

Logical Matrices

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

MATLAB Strings

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

Cell Arrays

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

Structures

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

Objects

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

Empty Arrays

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

Sparse Matrices

Sparse matrices have a different storage convention than full matrices 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 `j`th 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.

Using Data Types

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. The source file for this example is 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`.

For example, typing

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

produces this result

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

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

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

Building MEX-Files

This section covers the following topics:

- "Compiler Requirements"
- "Testing Your Configuration on UNIX"
- "Testing Your Configuration on Windows"
- "Specifying an Options File"

Compiler Requirements

Your installed version of MATLAB contains all the tools you need to work with the API. MATLAB includes a C compiler for the PC called Lcc, but does not include a Fortran compiler. If you choose to use your own C compiler, it must be an ANSI C 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).

MATLAB supports many compilers and provides preconfigured files, called options files, designed specifically for these compilers. The Options Files table lists all supported compilers and their corresponding options files. The purpose of supporting this large collection of compilers is to provide you with the flexibility to use the tool of your choice. However, in many cases, you simply can use the provided Lcc compiler with your C code to produce your applications.

The MathWorks also maintains a list of compilers supported by MATLAB at the following location on the web: `http://www.mathworks.com/support/tech-notes/1600/1601.shtml`.

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

The following sections contain configuration information for creating MEX-files on UNIX and Windows systems. More detailed information about the `mex` script is provided in "Custom Building MEX-Files" on page 3-18. In addition, there is a section on "Troubleshooting" on page 3-28, if you are having difficulties creating MEX-files.

Testing Your Configuration on UNIX

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`, included in the `<matlab>/extern/examples/mex` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system.

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 uses the system compiler to 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.

Selecting a Compiler

To change your default compiler, you select a different options file. You can do this anytime by using the command

```
mex -setup

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

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

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

The options files available for mex are:

  1: <matlab>/bin/gccopts.sh :
         Template Options file for building gcc MEXfiles

  2: <matlab>/bin/mexopts.sh :
         Template Options file for building MEXfiles using the
          system ANSI compiler

Enter the number of the options file to use as your default options
file:
```

Select the proper options file for your system by entering its number and pressing **Return**. If an options file doesn't exist in your MATLAB directory, the system displays a message stating that the options file is being copied to your user-specific `matlab` directory. If an options file already exists in your `matlab` directory, the system prompts you to overwrite it.

Note The `setup` option creates a user-specific `matlab` directory in your individual home directory and copies the appropriate options file to the directory. (If the directory already exists, a new one is not created.) This `matlab` directory is used for your individual options files only; each user can have his or her own default options files (other MATLAB products may place options files in this directory). Do not confuse these user-specific `matlab` directories with the system `matlab` directory, where MATLAB is installed. To see the name of this directory on your machine, use the MATLAB command `prefdir`.

Using the setup option resets your default compiler so that the new compiler is used every time you use the `mex` script.

Testing Your Configuration on Windows

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. To configure or change the options file at anytime, run

```
mex -setup
```

from either the MATLAB or DOS command prompt.

Selecting a Compiler

MATLAB includes a C compiler, Lcc, that you can use to create C MEX-files. The `mex` script will use the Lcc compiler automatically if you do not have a C or C++ compiler of your own already installed on your system and you try to compile a C MEX-file. Naturally, if you need to compile Fortran programs, you must supply your own supported Fortran compiler.

The `mex` script uses the filename extension to determine the type of compiler to use for creating your MEX-files. For example,

```
mex test1.f
```

would use your Fortran compiler and

```
mex test2.c
```

would use your C compiler.

On Systems without a Compiler. If you do not have your own C or C++ compiler on your system, the `mex` utility automatically configures itself for the included Lcc compiler. So, to create a C MEX-file on these systems, you can simply enter

```
mex filename.c
```

This simple method of creating MEX-files works for the majority of users.

If using the included Lcc compiler satisfies your needs, you can skip ahead in this section to “Building the MEX-File on Windows” on page 3-14.

On Systems with a Compiler. On systems where there is a C, C++, or Fortran compiler, you can select which compiler you want to use. Once you choose your compiler, that compiler becomes your default compiler and you no longer have to select one when you compile MEX-files. To select a compiler or change to existing default compiler, use `mex  setup`.

This example shows the process of setting your default compiler to the Microsoft Visual C++ Version 6.0 compiler.

```
mex -setup

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

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

Select a compiler:
[1] Compaq Visual Fortran version 6.6
[2] Lcc C version 2.4
[3] Microsoft Visual C/C++ version 6.0

[0] None

Compiler: 3

Your machine has a Microsoft Visual C/C++ compiler located at
D:\Applications\Microsoft Visual Studio. Do you want to use this
compiler [y]/n? y

Please verify your choices:

Compiler: Microsoft Visual C/C++ 6.0
Location: C:\Program Files\Microsoft Visual Studio

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

The default options file:
"C:\WINNT\Profiles\username\ApplicationData\MathWorks\MATLAB\R13
\mexopts.bat" is being updated from ...
```

If the specified compiler cannot be located, you are given the message:

```
The default location for compiler-name is directory-name,
but that directory does not exist on this machine.

Use directory-name anyway [y]/n?
```

Using the `setup` option sets your default compiler so that the new compiler is used every time you use the `mex` script.

Building the MEX-File on Windows

There is example C source code, `yprime.c`, and its Fortran counterpart, `yprimef.f` and `yprimefg.f`, included in the `<matlab>\extern\examples\mex` directory, where `<matlab>` represents the top-level directory where MATLAB is installed on your system.

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, switch to your Fortran compiler using `mex -setup`. Then, 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.

Specifying an Options File

You can use the `-f` option to specify an options file on either UNIX or Windows. To use the `-f` option, at the MATLAB prompt type

```
mex filename -f <optionsfile>
```

and specify the name of the options file along with its pathname. The Options Files table, below, 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 and UNIX)* 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. The Options Files table lists the compilers whose options files are included with this release of MATLAB.

Table 3-2: Options Files

Platform	Compiler	Options File
Windows	Borland C++, Version 5.0 & 5.2	`bccopts.bat`
	Borland C++Builder 3.0 (Borland C++, Version 5.3)	`bcc53opts.bat`
	Borland C++Builder 4.0 (Borland C++, Version 5.4)	`bcc54opts.bat`
	Borland C++Builder 5.0 (Borland C++, Version 5.5)	`bcc55opts.bat`
	Lcc C Compiler, bundled with MATLAB	`lccopts.bat`
	Microsoft C/C++, Version 5.0	`msvc50opts.bat`

Table 3-2: Options Files (Continued)

Platform	Compiler	Options File
	Microsoft C/C++, Version 6.0	`msvc60opts.bat`
	Watcom C/C++, Version 11	`wat11copts.bat`
	DIGITAL Visual Fortran, Version 5.0	`df50opts.bat`
	Compaq Visual Fortran, Version 6.1	`df61opts.bat`
	Compaq Visual Fortran, Version 6.6	`df66opts.bat`
	Borland C, Version 5.0 & 5.2, for Engine and MAT stand-alone programs	`bccengmatopts.bat`
	Borland C, Version 5.3, for Engine and MAT stand-alone programs	`bcc53engmatopts.bat`
	Borland C, Version 5.4, for Engine and MAT stand-alone programs	`bcc54engmatopts.bat`
	Borland C, Version 5.5, for Engine and MAT stand-alone programs	`bcc55engmatopts.bat`
	Lcc C compiler for Engine and MAT stand-alone programs,	`lccengmatopts.bat`
	Microsoft Visual C for Engine and MAT stand-alone programs, Version 5.0	`msvc50engmatopts.bat`
	Microsoft Visual C for Engine and MAT stand-alone programs, Version 6.0	`msvc60engmatopts.bat`
	Watcom C for Engine and MAT stand-alone programs, Version 11	`wat11engmatopts.bat`

Table 3-2: Options Files (Continued)

Platform	Compiler	Options File
	DIGITAL Visual Fortran for MAT stand-alone programs, Version 5.0	`df50engmatopts.bat`
	Compaq Visual Fortran for MAT stand-alone programs, Version 6.1	`df60engmatopts.bat`
UNIX	System ANSI Compiler	`mexopts.sh`
	GCC	`gccopts.sh`
	System ANSI Compiler for Engine stand-alone programs	`engopts.sh`
	System ANSI Compiler for MAT stand-alone programs	`matopts.sh`

An up-to-date list of options files is available from our FTP server, `ftp://ftp.mathworks.com/pub/tech-support/docexamples/apiguide/R12/bin`. For a list of all the compilers supported by MATLAB, access the MathWorks Technical Support Web site at `http://www.mathworks.com/support`.

Note The next section, "Custom Building MEX-Files" on page 3-18, contains specific information on how to modify options files for particular systems.

Custom Building MEX-Files

This section discusses in detail the process that the MEX-file build script uses. It covers the following topics:

- "Who Should Read this Chapter"
- "MEX Script Switches"
- "Default Options File on UNIX"
- "Default Options File on Windows"
- "Custom Building on UNIX"
- "Custom Building on Windows"

Who Should Read this Chapter

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).

MEX Script Switches

The `mex` script has a set of switches (also called options) that you can use to modify the link and compile stages. The MEX Script Switches table lists the available switches and their uses. Each switch is available on both UNIX and Windows unless otherwise noted.

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

Table 3-3: MEX Script Switches

Switch	Function
`@<rsp_file>`	Include the contents of the text file `<rsp_file>` as command line arguments to the `mex` script.
`-argcheck`	Perform argument checking on MATLAB API functions (C functions only).
`-c`	Compile only; do not link.
`-D<name>[#<def>]`	Define C preprocessor macro `<name>` [as having value `<def>`]. (Note: UNIX also allows `-D<name>[=<def>]`.)
`-f <file>`	Use `<file>` as the options file; `<file>` is a full pathname if it is not in current directory.
`-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.
`-inline`	Inlines matrix accessor functions (`mx*`). The generated MEX-function may not be compatible with future versions of MATLAB.
`-l<file>`	(UNIX) Link against library `lib<file>`.
`-L<pathname>`	(UNIX) Include `<pathname>` in the list of directories to search for libraries.

Table 3-3: MEX Script Switches (Continued)

Switch	Function
`<name>#<def>`	Override options file setting for variable `<name>`. This option is equivalent to `<ENV_VAR>#<val>`, which temporarily sets the environment variable `<ENV_VAR>` to `<val>` for the duration of the call to `mex`. `<val>` can refer to another environment variable by prepending the name of the variable with a `$`, e.g., `COMPFLAGS#"$COMPFLAGS -myswitch"`.
`<name>=<def>`	(UNIX) Override options file setting for variable `<name>`.
`-O`	Build an optimized executable.
`-outdir <name>`	Place all output files in directory `<name>`.
`-output <name>`	Create an executable named `<name>`. (An appropriate executable extension is automatically appended.)
`-setup`	Set up default options file. This switch should be the only argument passed.
`-U<name>`	Undefine C preprocessor macro `<name>`.
`-v`	Verbose; print all compiler and linker settings.
`-V5`	Compile MATLAB 5-compatible MEX-file.

Default Options File on UNIX

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

- The current directory
- The directory returned by the `prefdir` function
- The directory specified by `[matlabroot '/bin']`

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

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file in fullfile(matlabroot, 'bin', 'mexopts.sh'), or you can invoke the mex script in verbose mode (-v). Verbose mode will print the exact compiler options, prelink commands (if appropriate), and linker options used in the build process for each compiler. "Custom Building on UNIX" on page 3-22 gives an overview of the high-level build process.

Default Options File on Windows

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

- The current directory
- The user profile directory (returned by the prefdir function)
- The directory specified by [matlabroot '\bin\win32\mexopts']

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

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file, mexopts.bat, or you can invoke the mex script in verbose mode (-v). Verbose mode will print the exact compiler options, prelink commands, if appropriate, and linker options used in the build process for each compiler. "Custom Building on Windows" on page 3-24 gives an overview of the high-level build process.

The User Profile Directory

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

Custom Building on UNIX

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

Compile Stage

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`

Link Stage

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.

Build Options

For customizing the build process, you should modify the options file. The options file contains the compiler-specific flags corresponding to the general steps outlined above. The options file consists of a series of variable assignments; each variable represents a different logical piece of the build process. The options files provided with MATLAB are located in `<matlab>/bin`.

The section, "Default Options File on UNIX" on page 3-20, describes how the `mex` script looks for an options file.

To aid in providing flexibility, there are two sets of options in the options file that 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`	(none)

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

Custom Building on Windows

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

Compile Stage

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

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

Prelink Stage

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

- All MEX-files link against MATLAB only.
- MAT stand-alone programs link against `libmx.dll` (array access library), `libut.dll` (utility library), and `libmat.dll` (MAT-functions).
- Engine stand-alone programs link against `libmx.dll` (array access library), `libut.dll` (utility 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.

Link Stage

Finally, for the link stage, a `mex` options file must:

- Define the name of the linker in the `LINKER` environment variable.
- Define the `LINKFLAGS` environment variable that must contain:
  - The switch to create a DLL for MEX-files, or the switch to create an `exe` for stand-alone programs.
  - Export of the entry point to the MEX-file as `mexFunction` for C or `MEXFUNCTION@16` for DIGITAL Visual Fortran.
  - The import library (or libraries) 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's `.lib` file 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 Locate the `.LIB` files for the compiler version you are using under `matlabroot\extern\lib\win32\microsoft`. For example, for version 6.0, these files are in the `msvc60` subdirectory.

5 From this directory, add `libmx.lib`, `libmex.lib`, and `libmat.lib` 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**.

If you are using a compiler other than the Microsoft Visual C/C++ compiler, the process for building MEX files is similar to that described above. In step 4, locate the `.LIB` files for the compiler you are using in a subdirectory of `matlabroot\extern\lib\win32`. For example, for version 5.4 of the Borland C/C++ compiler, look in `matlabroot\extern\lib\win32\borland\bc54.`

Troubleshooting

This section explains how to troubleshoot some of the more common problems you may encounter. It addresses the following topics:

- “Configuration Issues”
- “Understanding MEX-File Problems”
- “Compiler and Platform-Specific Issues”
- “Memory Management Compatibility Issues”

Configuration Issues

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 with certain C compilers.

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.

Internal Error When Using mex -setup (PC).

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

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

then you need to disable your antivirus software temporarily and reenter the command. After you have successfully run the mex operation, you can re-enable your antivirus software.

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

General Configuration Problem

Make sure you followed the configuration steps for your platform described in this chapter. Also, refer to "Custom Building MEX-Files" on page 3-18 for additional information.

Understanding MEX-File Problems

This section contains information regarding common problems that occur when creating MEX-files. Use the figure, below, to help isolate these problems.

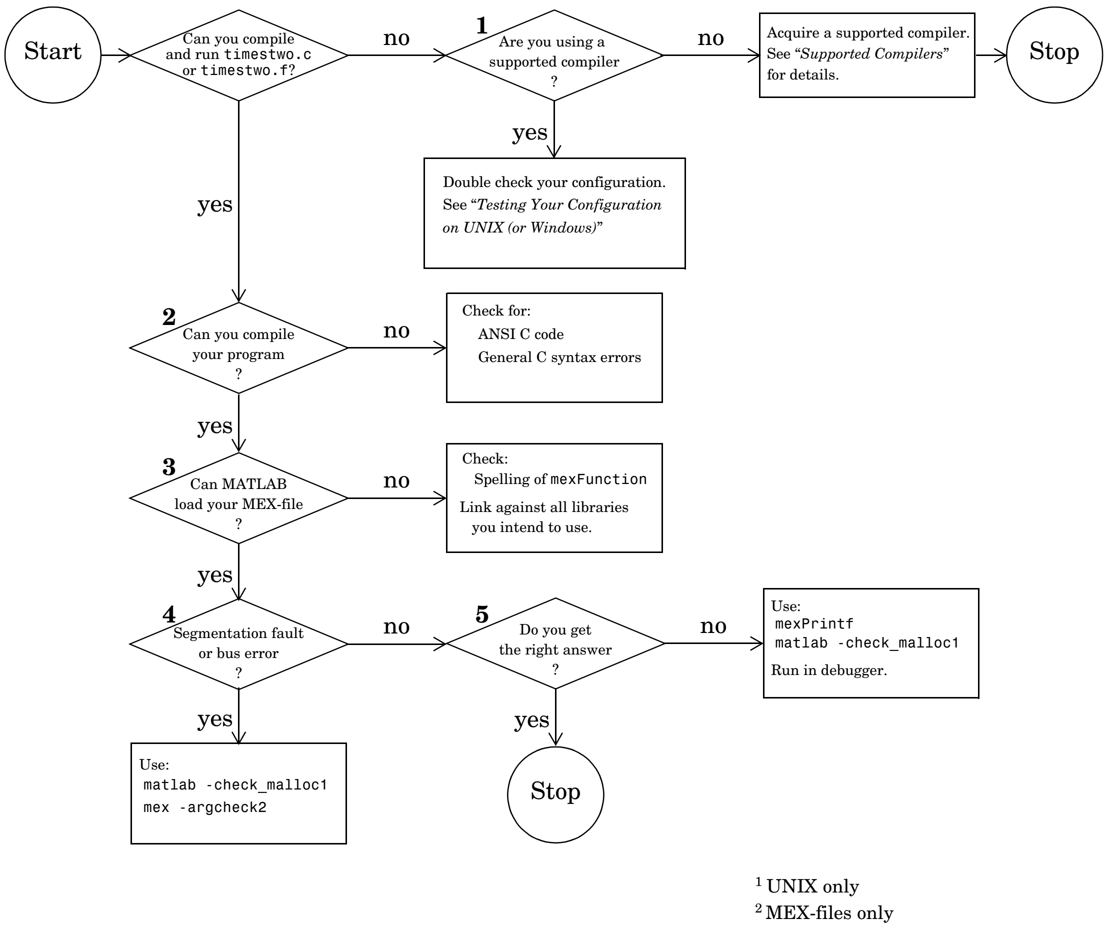

Start
Can you compile and run timestwo.c or timestwo.f?
no
1 Are you using a supported compiler ?
no
Acquire a supported compiler. See "Supported Compilers" for details.
Stop
yes
Double check your configuration. See "Testing Your Configuration on UNIX (or Windows)"
yes
2 Can you compile your program ?
no
Check for: ANSI C code General C syntax errors
yes
3 Can MATLAB load your MEX-file ?
no
Check: Spelling of mexFunction Link against all libraries you intend to use.
yes
4 Segmentation fault or bus error ?
no
5 Do you get the right answer ?
no
Use: mexPrintf matlab -check_malloc1 Run in debugger.
yes
Use: matlab -check_malloc1 mex -argcheck2
yes
Stop
1 UNIX only
2 MEX-files only

Figure 3-1: Troubleshooting MEX-File Creation Problems

Problems 1 through 5 refer to specific sections of the previous flowchart. For additional suggestions on resolving MEX build problems, access the MathWorks Technical Support Web site at `http://www.mathworks.com/support`.

Problem 1 - Compiling a MathWorks Program Fails

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" on page 3-9 for information on selecting the appropriate options file or, if necessary, obtain an ANSI C compiler.

Problem 2 - Compiling Your Own Program Fails

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 - MEX-File Load Errors

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 - Segmentation Fault or Bus Error

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 - Program Generates Incorrect Results

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

Another 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.

Compiler and Platform-Specific Issues

This section refers to situations specific to particular compilers and platforms.

MEX-Files Created in Watcom IDE

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

Memory Management Compatibility Issues

MATLAB now 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[]`). MATLAB issues a warning when it 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 “Memory Management” on page 4-37 in Creating C Language MEX-Files.

Note Currently, the following 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.

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.
```

Example That Causes Warning

In the following example, `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.

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

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

In the following example, 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).

```
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 */
```

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`, specifying memory that was not allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc` as the intended data block (second argument), then when the MEX-file returns, MATLAB 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.

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

Solution

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

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

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 `mexGetVariablePtr` 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.)

Additional Information

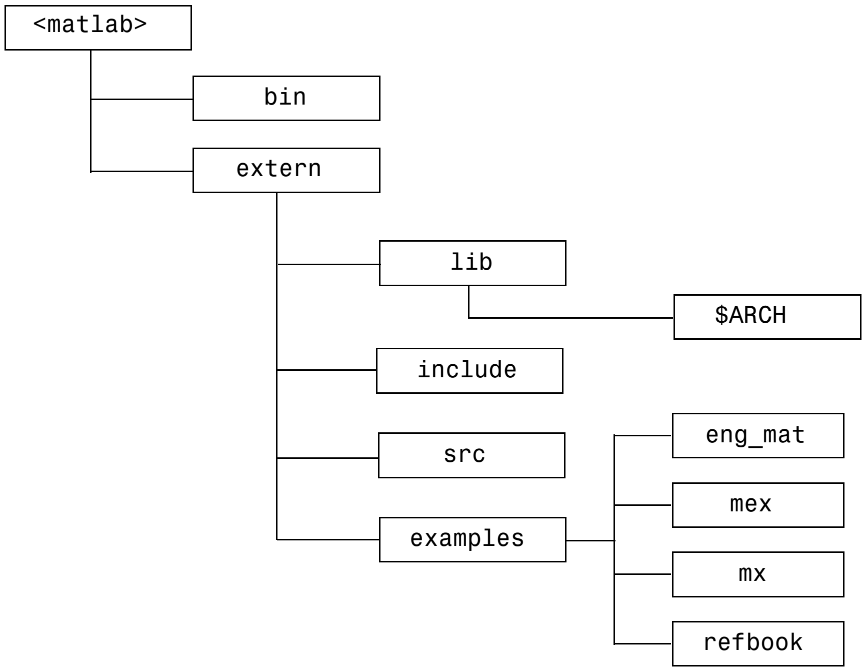
The following sections describe how to find additional information and assistance in building your applications. It covers the following topics:

- "Files and Directories - UNIX Systems"
- "Files and Directories - Windows Systems"
- "Examples"
- "Technical Support"

Files and Directories - UNIX Systems

This section describes the directory organization and purpose of the files associated with the MATLAB API on UNIX systems.

The following 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.
`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. This table lists the options files.

Table 3-4: Preconfigured Options Files

Options File	Description
`engopts.sh`	Used with the `mex` script and the system C or Fortran compiler to compile engine applications
`gccopts.sh`	Used with the `mex` script and the GNU C (`gcc`) compiler to compile MEX-files
`matopts.sh`	Used with the `mex` script and the system C or Fortran compiler to compile MAT-file applications
`mexopts.sh`	Used with the `mex` script and the system ANSI C or Fortran compiler to compile MEX-files

<matlab>/extern/lib/$ARCH

The `<matlab>/extern/lib/`*`$ARCH`* directory contains libraries, where *`$ARCH`* specifies a particular UNIX platform. 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.

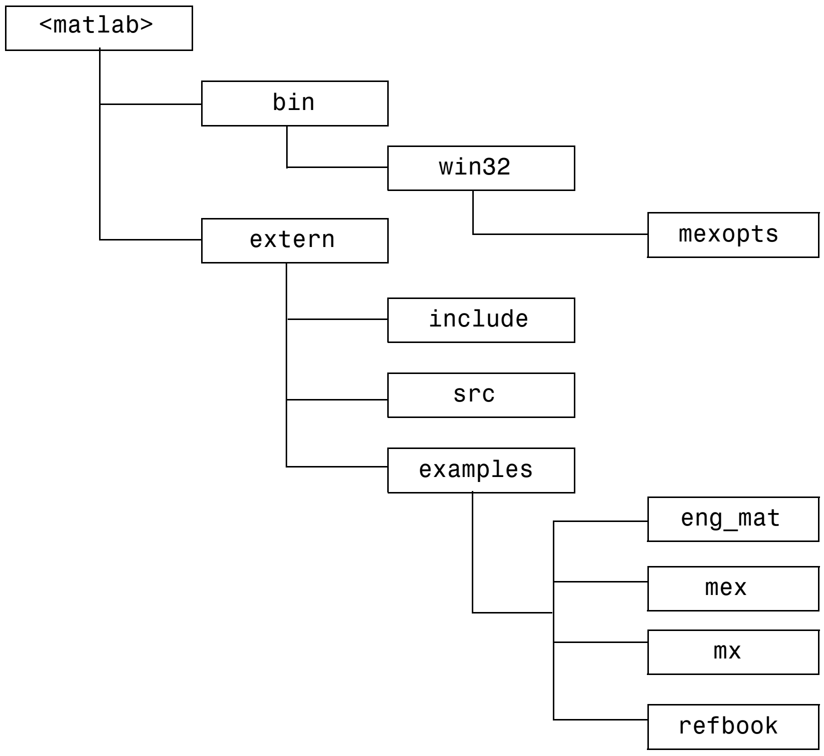
<matlab>/extern/src

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.

Files and Directories - Windows Systems

This section describes the directory organization and purpose of the files associated with the MATLAB API on Microsoft Windows systems.

The following 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\win32

The `<matlab>\bin\win32` directory contains the `mex.bat` batch file that builds C and Fortran files into MEX-files. Also, this directory contains `mex.pl`, which is a Perl script used by `mex.bat`.

<matlab>\bin\win32\mexopts

The `<matlab>\bin\win32\mexopts` directory contains the preconfigured options files that the `mex` script uses with particular compilers. See Table 3-2, Options Files, on page 3-15 for a complete list of the options files.

<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.

Examples

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

Examples from the Text

The `refbook` subdirectory in the `extern/examples` directory contains the MEX-file examples (C and Fortran) that are used in this book, *External Interfaces*.

You can find the most recent versions of these examples using the anonymous FTP server locations

```
ftp://ftp.mathworks.com/pub/tech-support/docexamples/apiguide/R1
2/refbook
```

MEX Reference Examples

The `mex` subdirectory of `/extern/examples` directory contains MEX-file examples. It includes the examples described in the online External Interfaces/API reference pages for MEX interface functions (the functions beginning with the `mex` prefix).

You can find the most recent versions of these examples using the anonymous FTP server location

```
ftp://ftp.mathworks.com/pub/tech-support/docexamples/apiguide/R1
2/mex
```

MX Examples

The `mx` subdirectory of `extern/examples` 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.

You can find the most recent versions of these examples using the anonymous FTP server location

```
ftp://ftp.mathworks.com/pub/tech-support/docexamples/apiguide/R1
2/mx
```

Engine and MAT Examples

The eng_mat subdirectory in the extern/examples directory contains the MEX-file examples (C and Fortran) for using the MATLAB engine facility, as well as examples for reading and writing MATLAB data files (MAT-files). These examples are all stand-alone programs.

You can find the most recent versions of these examples using the anonymous FTP server locations

```
ftp://ftp.mathworks.com/pub/tech-support/docexamples/apiguide/R1
2/eng_mat
```

Technical Support

The MathWorks provides additional Technical Support through its web site. A few of the services provided are as follows:

- Solution Search Engine

This knowledge base on our web site includes thousands of solutions and links to Technical Notes and is updated several times each week.

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

- Technical Notes

Technical notes are written by our Technical Support staff to address commonly asked questions.

```
http://www.mathworks.com/support/tech-notes/list_all.shtml
```

Creating C Language MEX-Files

This chapter describes how to write MEX-files in the C programming language. It discusses the MEX-file itself, how these C language files interact with MATLAB, how to pass and manipulate arguments of different data types, how to debug your MEX-file programs, and several other, more advanced topics.

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.

The Components of a C 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.

The following C MEX Cycle figure shows how inputs enter a MEX-file, what functions the gateway routine performs, and how outputs return to MATLAB.

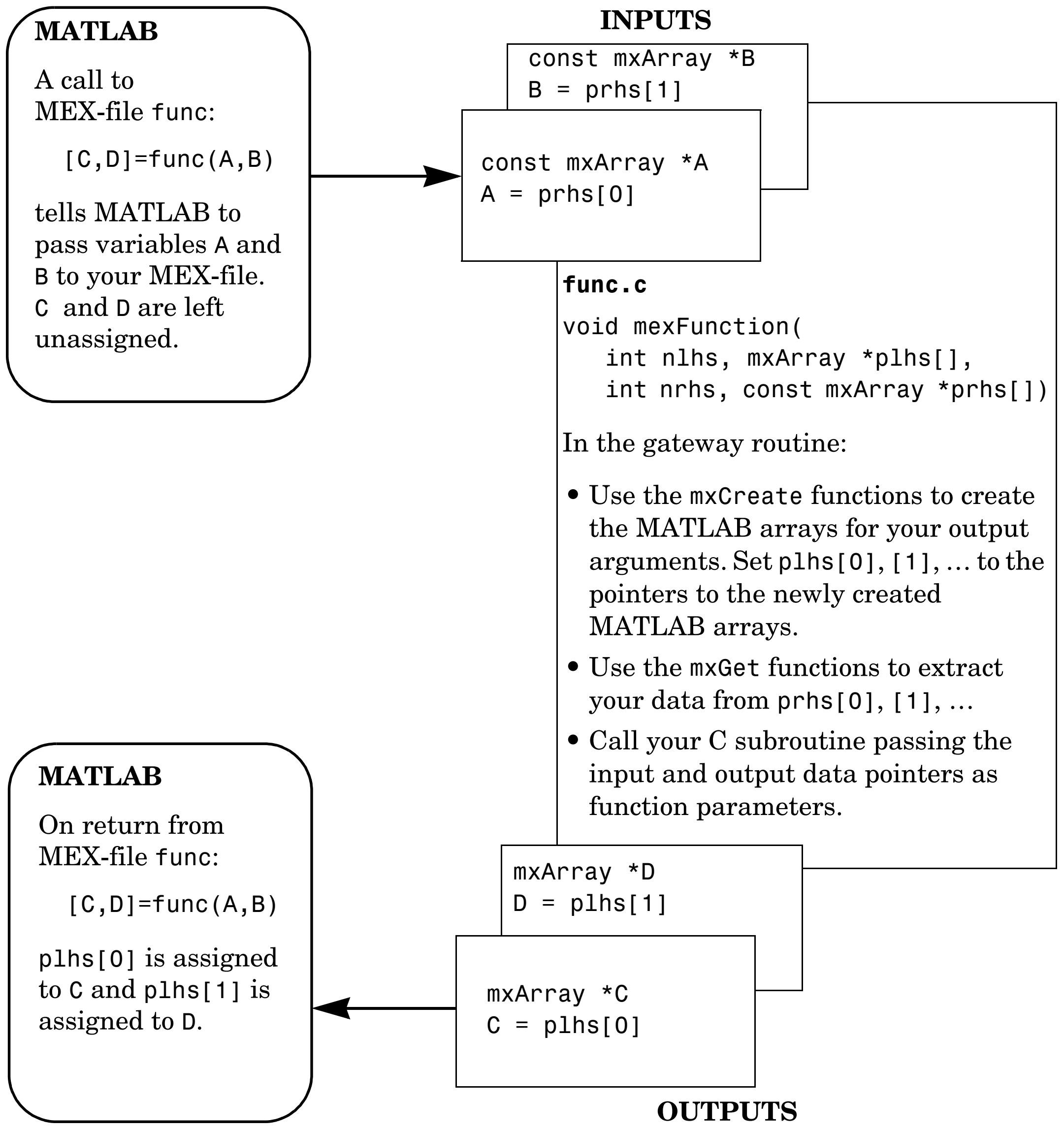

MATLAB
A call to
MEX-file func:
[C,D]=func(A,B)
tells MATLAB to pass variables A and B to your MEX-file. C and D are left unassigned.
INPUTS
const mxArray *B
B = prhs[1]
const mxArray *A
A = prhs[0]
func.c
void mexFunction(
int nlhs, mxArray *plhs[],
int nrhs, const mxArray *prhs[])
In the gateway routine:
• Use the mxCreate functions to create the MATLAB arrays for your output arguments. Set plhs[0], [1], … to the pointers to the newly created MATLAB arrays.
• Use the mxGet functions to extract your data from prhs[0], [1], …
• Call your C subroutine passing the input and output data pointers as function parameters.
MATLAB
On return from
MEX-file func:
[C,D]=func(A,B)
plhs[0] is assigned to C and plhs[1] is assigned to D.
mxArray *D
D = plhs[1]
mxArray *C
C = plhs[0]
OUTPUTS

Figure 4-1: 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, )
```

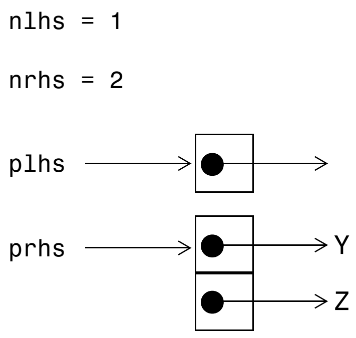
where the ellipsis ( ) denotes additional terms of the same format. The `a,b,c,` are left-hand arguments and the `d,e,f,` are right-hand arguments.

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.



`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 following sections include information and examples describing how to pass and manipulate the different data types when working with MEX-files. These topics include

- "A First Example — Passing a Scalar"
- "Passing Strings"
- "Passing Two or More Inputs or Outputs"
- "Passing Structures and Cell Arrays"
- "Handling Complex Data"
- "Handling 8-,16-, and 32-Bit Data"
- "Manipulating Multidimensional Numerical Arrays"
- "Handling Sparse Arrays"
- "Calling Functions from C MEX-Files"

The MATLAB API provides a full set of routines that handle the various data types supported by MATLAB. For each data type there is a specific set of functions that you can use for data manipulation. The first example discusses the simple case of doubling a scalar. After that, the examples discuss how to pass in, manipulate, and pass back various data types, and how to handle multiple inputs and outputs. Finally, the sections discuss passing and manipulating various MATLAB data types.

Note You can find the most recent versions of the example programs at the anonymous FTP server

```
ftp://ftp.mathworks.com/pub/tech-support/docexamples/apiguide/R12/
refbook
```

A First Example — Passing a Scalar

Let's look at a simple example of C code and its MEX-file equivalent. Here is a C computational function that takes a scalar and doubles it.

```
#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.

```
/*
 * =============================================================
 * 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) 1984-2000 The MathWorks, Inc.
 * =============================================================
 */

/* $Revision: 1.8 $ */

#include "mex.h"

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).

```
/*
 * =============================================================
 * 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) 1984-2000 The MathWorks, Inc.
 * =============================================================
 */

/* $Revision: 1.5 $ */

#include "mex.h"

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.

Passing 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.

```
/*
 * ================================================================
 * 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) 1984-2000 The MathWorks, Inc.
 * ===============================================================
 */
```

```
/* $Revision: 1.10 $ */

#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 the MATLAB memory manager and initializes it to zero. You must use `mxCalloc` in any situation where C would require the use of `calloc`. The same is true for `mxMalloc` and `mxRealloc`; use `mxMalloc` in any situation where C would require the use of `malloc` and use `mxRealloc` where C would require `realloc`.

Note MATLAB automatically frees up memory allocated with the `mx` allocation routines (`mxCalloc`, `mxMalloc`, `mxRealloc`) upon exiting your MEX-file. If you don't want this to happen, use the API function `mexMakeMemoryPersistent`.

Below is the gateway routine 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. */
  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 routine 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.

```
/*
 * =============================================================
 * 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) 1984-2000 The MathWorks, Inc.
 * =============================================================
 */

/* $Revision: 1.10 $ */

#include "mex.h"

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 routine */
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. */
  /* NOTE: You do not need an else statement when using
     mexErrMsgTxt within an if statement. It will never
     get to the else statement if mexErrMsgTxt is executed.
     (mexErrMsgTxt breaks you out of the MEX-file.)
  */
  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 (!mxIsDouble(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`.

Passing Structures and Cell Arrays

Passing structures and cell arrays into MEX-files is just like passing any other data types, except the data itself is of type `mxArray`. In practice, this means that `mxGetField` (for structures) and `mxGetCell` (for cell arrays) return pointers of type `mxArray`. You can then treat the pointers like any other pointers of type `mxArray`, but if you want to pass the data contained in the `mxArray` to a C routine, you must use an API function such as `mxGetData` to access it.

This example takes an m-by-n structure matrix as input and returns a new 1-by-1 structure that contains these fields:

- String input generates an m-by-n cell array
- Numeric input (noncomplex, scalar values) generates an m-by-n vector of numbers with the same class ID as the input, for example `int`, `double`, and so on.

```
/*
 * =============================================================
 * phonebook.c
 * Example for illustrating how to manipulate structure and cell
 * array
 *
 * Takes a (MxN) structure matrix and returns a new structure
 * (1x1) 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) 1984-2000 The MathWorks, Inc.
 * =============================================================
 */

/* $Revision: 1.6 $ */

#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; 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)) ||
          mxIsSparse(tmp)) {
        mexPrintf("%s%d\t%s%d\n",
            "FIELD:", ifield+1, "STRUCT INDEX :", jstruct+1);
        mexErrMsgTxt("Above field must have either "
            "string or numeric non-sparse 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. The MATLAB API provides two functions, `mxGetPr` and `mxGetPi`, that return pointers (of type `double *`) to the real and imaginary parts of your data.

This example takes two complex row vectors and convolves them.

```
/*
 * =============================================================
 * 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) 1984-2000 The MathWorks, Inc.
 * =============================================================
 */

/* $Revision: 1.8 $ */
```

```
#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 routine 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 API provides a set of functions that support these data types. The API function `mxCreateNumericArray` constructs an unpopulated N-dimensional numeric array with a specified data size. Refer to the entry for `mxClassID` in the online reference pages for a discussion of how the MATLAB API represents these data types.

Once you have created an unpopulated MATLAB array of a specified data type, you can access the data using `mxGetData` and `mxGetImagData`. These two functions return pointers to the real and imaginary data. You can perform arithmetic on data of 8-, 16- or 32-bit precision in MEX-files and return the result to MATLAB, which will recognize the correct data class.

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

```
/*
 * =============================================================
 * 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) 1984-2000 The MathWorks, Inc.
 * =============================================================
 */

/* $Revision: 1.9 $ */

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

#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 gateway routine */
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 *)mxGetData(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.

Manipulating Multidimensional Numerical Arrays

You can manipulate multidimensional numerical arrays by using `mxGetData` and `mxGetImagData` to return pointers to the real and imaginary parts of the data stored in the original multidimensional array. 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) 1984-2000 by The MathWorks, Inc.
 * ===============================================================
 */

/* $Revision: 1.5 $ */

#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 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 section, "The MATLAB Array" on page 3-4.

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
 * non-sparse 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) 1984-2000 The MathWorks, Inc.
 * =============================================================
 */

/* $Revision: 1.5 $ */

#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 defined(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 j,k,m,n,nzmax,*irs,*jcs,cmplx,isfull;
  double *pr,*pi,*si,*sr;
  double percent_sparse;

  /* 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.
 */

percent_sparse = 0.2;
nzmax = (int)ceil((double)m*(double)n*percent_sparse);

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

/* Copy nonzeros. */
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, increase
       * percent_sparse by 10%, recalculate nzmax, and augment
       * the sparse array.
       */
      if (k >= nzmax) {
        int oldnzmax = nzmax;
        percent_sparse += 0.1;
        nzmax = (int)ceil((double)m*(double)n*percent_sparse);
```

```
              /* Make sure nzmax increases atleast by 1. */
              if (oldnzmax == nzmax)
                nzmax++;

              mxSetNzmax(plhs[0], nzmax);
              mxSetPr(plhs[0], mxRealloc(sr, nzmax*sizeof(double)));
              if (si != NULL)
              mxSetPi(plhs[0], mxRealloc(si, nzmax*sizeof(double)));
              mxSetIr(plhs[0], mxRealloc(irs, nzmax*sizeof(int)));

              sr  = mxGetPr(plhs[0]);
              si  = mxGetPi(plhs[0]);
              irs = mxGetIr(plhs[0]);
            }
            sr[k] = pr[i];
            if (cmplx) {
              si[k] = pi[i];
            }
            irs[k] = i;
            k++;
          }
        }
        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 Functions from C MEX-Files

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.

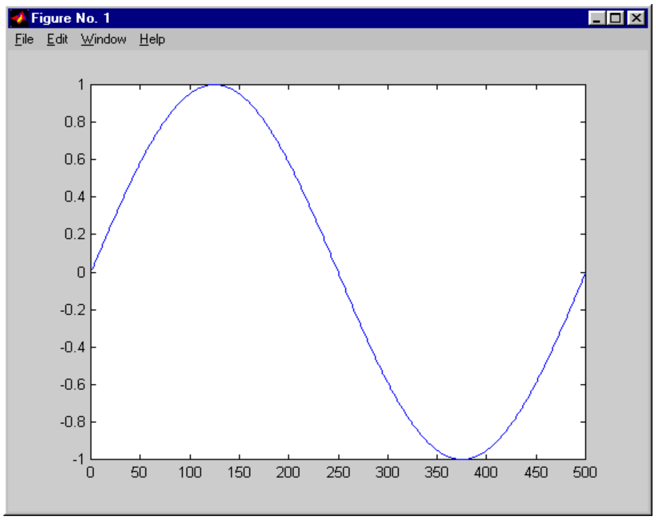
```
/*
 * =============================================================
 * sincall.c
 *
 * Example for illustrating how to use mexCallMATLAB
 *
 * Creates an mxArray and passes its associated pointers (in
 * this demo, only pointer to its real part, pointer to number of
 * rows, pointer to number of 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) 1984-2000 The MathWorks, Inc.
 * =============================================================
 */

/* $Revision: 1.4 $ */

#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);
}

/* The gateway routine */
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`. See the example below.

The following example creates an M-file that returns two variables but only assigns one of them a value.

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

MATLAB displays the following warning message.

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

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