## Introducing the MATLAB API

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1-2</td>
</tr>
<tr>
<td>How This Book Is Organized</td>
<td>1-2</td>
</tr>
<tr>
<td>The API Documentation Set</td>
<td>1-3</td>
</tr>
<tr>
<td>Dynamically Linked Subroutines: MEX-Files</td>
<td>1-4</td>
</tr>
<tr>
<td>Using MEX-Files</td>
<td>1-4</td>
</tr>
<tr>
<td>The MATLAB Array</td>
<td>1-5</td>
</tr>
<tr>
<td>The Distinction Between the mx and mex Prefixes</td>
<td>1-7</td>
</tr>
<tr>
<td>Data Types in MATLAB</td>
<td>1-7</td>
</tr>
<tr>
<td>Getting Started</td>
<td>1-9</td>
</tr>
<tr>
<td>Testing Your Configuration</td>
<td>1-9</td>
</tr>
<tr>
<td>On Windows</td>
<td>1-9</td>
</tr>
<tr>
<td>On UNIX</td>
<td>1-10</td>
</tr>
<tr>
<td>On Macintosh</td>
<td>1-10</td>
</tr>
<tr>
<td>Specifying an Options File</td>
<td>1-14</td>
</tr>
<tr>
<td>Troubleshooting Your Configuration</td>
<td>1-16</td>
</tr>
<tr>
<td>Search Path Problem</td>
<td>1-16</td>
</tr>
<tr>
<td>Compiler Problem</td>
<td>1-16</td>
</tr>
<tr>
<td>General Configuration Problem</td>
<td>1-16</td>
</tr>
</tbody>
</table>

## Creating C Language MEX-Files

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C MEX-Files</td>
<td>2-2</td>
</tr>
<tr>
<td>Directory Organization</td>
<td>2-2</td>
</tr>
<tr>
<td>The Parts of a MEX-File</td>
<td>2-2</td>
</tr>
<tr>
<td>Required Arguments to a MEX-File</td>
<td>2-5</td>
</tr>
<tr>
<td>Examples of C MEX-Files</td>
<td>2-7</td>
</tr>
<tr>
<td>A First Example</td>
<td>2-7</td>
</tr>
</tbody>
</table>
Creating Fortran MEX-Files

Fortran MEX-Files .......................................................... 3-2
  Directory Organization .................................................. 3-2
  MEX-Files and Data Types .............................................. 3-2
  The Components of a Fortran MEX-File .............................. 3-3
    The Pointer Concept .................................................. 3-5
    The Gateway Routine .................................................. 3-6
    The %val Construct .................................................... 3-7

Examples of Fortran MEX-Files ........................................... 3-9
  A First Example—Passing a Scalar ................................. 3-9
  Passing Strings .......................................................... 3-11
  Passing Arrays of Strings ............................................. 3-13
  Passing Matrices ......................................................... 3-14
  Passing Two or More Inputs or Outputs .............................. 3-17
### Data Export and Import

**Using MAT-Files** ............................................. 4-2
- Importing Data to MATLAB .................................. 4-2
- Exporting Data from MATLAB ............................... 4-3
- Exchanging Data Files Between Platforms .............. 4-4
- Reading and Writing MAT-Files ......................... 4-5
  - MAT-File Subroutine Library ......................... 4-5
- Directory Organization ................................. 4-7

**Examples of MAT-Files** ..................................... 4-11
- Writing a MAT-File ....................................... 4-11
- Reading a MAT-File ..................................... 4-13

**Compiling and Linking MAT-File Programs** ............ 4-15
- On UNIX ........................................... 4-15
- Windows ........................................ 4-15
Using the MATLAB Engine

Interprocess Communication: The MATLAB Engine .................. 5-2
  Example: Calling the MATLAB Engine from a
  C Program .............................................. 5-3

Compiling and Linking Engine Programs .............................. 5-5
  Windows .............................................. 5-5
  UNIX .................................................. 5-5
  Example: Calling the MATLAB Engine from a
  Fortran Program ....................................... 5-7

ActiveX Automation for Windows ...................................... 5-8
  MATLAB ActiveX Automation Methods ......................... 5-9
  Additional ActiveX Information .............................. 5-12
    Launching the MATLAB ActiveX Server ...................... 5-12
    Establishing Multiple Connections ......................... 5-12
    DCOM: Using MATLAB ActiveX over a
    Network Connection .................................... 5-13

Dynamic Data Exchange (DDE) ....................................... 5-14
  DDE Concepts and Terminology ............................... 5-14
  The Service Name ...................................... 5-15
  The Topic ............................................. 5-15
  The Item ............................................. 5-15
  Clipboard Formats ...................................... 5-15
Introducing the MATLAB API

1-2 Introduction
1-2 How This Book Is Organized
1-3 The API Documentation Set

1-4 Dynamically Linked Subroutines: MEX-Files
1-4 Using MEX-Files
1-5 The MATLAB Array
1-7 Data Types in MATLAB

1-9 Getting Started
1-9 Testing Your Configuration
1-14 Specifying an Options File
1-16 Troubleshooting Your Configuration
Introduction

Although MATLAB® is a complete, self-contained environment for programming and working with data, it is often useful to interact with data and programs external to MATLAB. MATLAB provides an Application Program Interface (API) to support these external interfaces. The functions supported by the API include:

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

How This Book Is Organized

Chapter 1 discusses MEX-files, which enable you to call your own C, C++, or Fortran subroutines directly from MATLAB. It also provides basic information to get you up and running so that you can configure your system to build MEX-functions.

Chapters 2 and 3 contain C and Fortran examples that explain how to create MEX-files on UNIX, Macintosh, and PC platforms.

Chapter 4 continues with a discussion of techniques for importing and exporting data to and from the MATLAB environment. The most important technique is MAT-files – the files MATLAB uses for saving data to a disk. MAT-files offer a simple and convenient mechanism for transporting your data between different platforms. They also enable you to import and export your MATLAB data to and from other MATLAB stand-alone applications. To simplify the use of MAT-files with other applications, a library of access routines is provided that makes it very easy to read and write MAT-files using your own C or Fortran programs.

Chapter 5 discusses the MATLAB Engine, which enables you to set up client/server relationships between MATLAB and other software programs, such as Excel.

Chapter 6 focuses on platform-specific issues, such as directory organization and setting correct paths if you install a new compiler after having installed MATLAB. There is also a discussion of how to build MEX, MAT, and engine routines outside of MATLAB by using Integrated Development Environments.
(IDEs). In addition, Chapter 6 contains information on troubleshooting MEX-file creation.

The API Documentation Set
This book, the Application Program Interface Guide, contains configuration information and tutorials for using the MATLAB API. The complete set of reference documentation for all the API-related functions is provided online, and can be accessed from the MATLAB Help Desk by typing `helpdesk` at the MATLAB prompt. From the Help Desk, you can also access online (pdf) versions of the Application Program Interface Guide and the Application Program Interface Reference. The online version of the Application Program Interface Reference is the complete set of API reference pages in a book format.
Dynamically Linked Subroutines: MEX-Files

You can call your own C and 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 Fortran and C 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 dynamically linked 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>
<thead>
<tr>
<th>Platform</th>
<th>MEX-File Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun OS 4.x</td>
<td>mex4</td>
</tr>
<tr>
<td>HP 9000/series 700</td>
<td>mexhp7</td>
</tr>
<tr>
<td>Alpha</td>
<td>mexds</td>
</tr>
<tr>
<td>SGI</td>
<td>mexsg</td>
</tr>
<tr>
<td>SGI 64</td>
<td>mexsg64</td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td>mexrs6</td>
</tr>
<tr>
<td>Linux</td>
<td>mexlx</td>
</tr>
<tr>
<td>Solaris</td>
<td>mexsol</td>
</tr>
<tr>
<td>Windows</td>
<td>dll</td>
</tr>
<tr>
<td>Macintosh</td>
<td>mex</td>
</tr>
</tbody>
</table>

You can call MEX-files exactly as you would call any M-function. For example, a MEX-file called \texttt{ode23.mex} on your disk in the MATLAB toolbox integrates ordinary differential equations. This is a fast version of the M-file \texttt{ode23.m}. If you invoke the function \texttt{ode23} from inside MATLAB, the interpreter looks through the list of directories on MATLAB’s search path. It scans each directory looking for the first occurrence of a file named \texttt{ode23} with the corresponding filename extension from the table or \texttt{.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 \texttt{.m} file.

**The MATLAB Array**

Before you can program MEX-files, you must understand how MATLAB represents the many data types it supports. The MATLAB language works with only a single object type: the MATLAB array. All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, and structures are
stored as MATLAB arrays. The `mxArray` declaration corresponds to the internal data structure that MATLAB uses to represent arrays. The MATLAB array is the C language definition of a MATLAB variable. This `mxArray` structure contains, among other things:

- The MATLAB variable's name
- Its dimensions
- Its type
- Whether the variable is real or complex. If the variable contains complex numbers as elements, the MATLAB array includes vectors containing the real and imaginary parts
- Whether the variable is a sparse matrix

Matrices, or m-by-n arrays, that are not sparse are called full. In the case of a full matrix, the `mxArray` structure contains parameters called `pr` and `pi`. `pr` contains the real part of the matrix data; `pi` contains the imaginary data, if there is any. Both `pr` and `pi` are one-dimensional arrays of double-precision numbers. The elements of the matrix are stored in these arrays columnwise. This is how Fortran stores matrices; MATLAB uses this convention because it was originally written in Fortran.

Sparse matrices have a different storage convention in MATLAB. The parameters `pr` and `pi` are still arrays of double-precision numbers, but there are now three new 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.
The Distinction Between the mx and mex Prefixes

Rather than directly manipulating the MATLAB array data structure from C (which is possible) or from Fortran (which is not), the MEX-file interface library provides a set of access methods (routines) for manipulating MATLAB arrays. These subroutines, which are fully documented in the online reference pages, always start with the letters \texttt{mx}. This is shorthand for \texttt{mxArray} and it means that the subroutine enables you to access and/or manipulate some of the information in the MATLAB array. For example, \texttt{mxGetPi} retrieves the pointer to the imaginary data inside the array. The only exceptions to this principle are \texttt{mxGetEps}, \texttt{mxGetNan}, and \texttt{mxGetInf}, which do not access or manipulate the MATLAB array.

There is also a set of routines that begin with the prefix \texttt{mex}. Any routine that begins with \texttt{mex} performs some operation back in the MATLAB environment. For example, the \texttt{mexPrintf} prints a string at the MATLAB prompt, even though the call came from within a MEX-file. No function with the \texttt{mex} prefix directly accesses or manipulates information in the \texttt{mxArray} data structure.

Data Types in MATLAB

You can write MEX-files in C that accept any data type supported by MATLAB. In Fortran, only double-precision n-by-m arrays and strings are supported. You can treat C and Fortran MEX-files, once compiled, exactly like M-functions.

Let’s look at an example of a MEX-file. Included with MATLAB is a MEX-file called \texttt{explore}, which is a function that tells you what data type the input variable is. For example, typing

\begin{verbatim}
x = 2;
explore(x);
\end{verbatim}

produces this result:

\begin{verbatim}
-------------------------------------------
Name: x
Dimensions: 1x1
Class Name: double
-------------------------------------------
(1,1) = 2
\end{verbatim}
explore accepts any data type, including strings, structures, cell-arrays, and all other MATLAB data types.

\[ y = 'hello world' \]
\[ \text{explore}(y); \]

```
Name: y
Dimensions: 1x1
Class Name: char
```

\( (1,1) \) hello world

Try constructing any data type you want and then use `explore` to verify the data type.
Getting Started

Your installed version of MATLAB contains all the tools you need to work with the API, except a C or Fortran compiler. The API supports many compilers, and provides options files designed specifically for these compilers. Chapter 6, “System Setup,” provides detailed information on the compilers, options files, and customization.

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

Additional information about the `mex` script is provided in “Custom Building of MEX-Files” in Chapter 6. You’ll also find troubleshooting information if you are having difficulties creating MEX-files in the “Troubleshooting MEX-File Creation” section.

Testing Your Configuration

The quickest way to see 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.for` and `yprimefg.for` (Windows and Macintosh) and `yprime.f` and `yprimefg.F` (UNIX), included in the `examples/mex` directory.

The following sections contain configuration information for creating MEX-files on Windows, UNIX, and Macintosh systems. If, after following the instructions, you have difficulty creating MEX-files, refer to Chapter 6 for additional troubleshooting information.

On Windows

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

```bash
mex yprime.c
```
Introducing the MATLAB API

This should create the MEX-file called \texttt{yprime} with the .DLL extension, which corresponds to the Windows platform. You can now call \texttt{yprime} as if it were an M-function:

\begin{verbatim}
  yprime(1,1:4)
  ans =
2.0000 8.9685 4.0000 -1.0947
\end{verbatim}

To try the Fortran version of the sample program, at the MATLAB prompt, type:

\begin{verbatim}
  mex yprimef.for yprimefg.for
\end{verbatim}

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

\textbf{On UNIX}

To compile and link the example source files, \texttt{yprime.c} or \texttt{yprime.f} and \texttt{yprimefg.F}, on UNIX, you must first copy the file(s) to a local directory, and then change directory (\texttt{cd}) to that local directory. After performing these steps, you can follow the instructions provided in the “On Windows” section to test your configuration. On UNIX, the MEX-file is created with the platform-specific extension, as shown on page 1-5.

\textbf{On Macintosh}

Before you can build MEX-files on Macintosh systems, you must perform several configuration steps. On the Macintosh, the \texttt{mex} script uses an options file, \texttt{mexopts}, that is located in the \texttt{<MATLAB>:extern:scripts:} folder. MATLAB includes three preconfigured \texttt{mexopts} files in the \texttt{<MATLAB>:extern:scripts:} folder, and you must configure your system to use the one you need.

<table>
<thead>
<tr>
<th>\textbf{mexopts File}</th>
<th>\textbf{Compiler}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mexopts.cw}</td>
<td>Metrowerks CodeWarrior C</td>
</tr>
<tr>
<td>\texttt{mexopts.mpwc}</td>
<td>MPW MrC/SC</td>
</tr>
<tr>
<td>\texttt{mexopts.ls}</td>
<td>Language Systems Fortran</td>
</tr>
</tbody>
</table>
To select one of these preconfigured files as a default `mexopts` file:

1. Make an alias of or copy the file.
2. Rename the alias/copy to the name `mexopts`.

The `mex` script works with all three compilers listed above. However, there are some additional steps you must follow if you are using MPW or Language Systems (LS) Fortran. See the section, “Special Considerations for MPW and LS Fortran Users,” later in this chapter for more information.

The first time you run the `mex` script, dialogs may appear that ask you to find and select either the CodeWarrior IDE application or the ToolServer application. This information is saved in the `<MATLAB>:extern:scripts:` folder, so be sure you have write privileges enabled for that folder.

To compile and link the example source file, `yprime.c`, on the Macintosh, at the MATLAB prompt, type:

```matlab
mex yprime.c
```

This should create the MEX-file called `yprime` with the `.mex` extension, which corresponds to the Macintosh platform. You can now call `yprime` as if it were an M-function:

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

To try the Fortran version of the sample program, `yprimef.for` and `yprimefg.for`, at the MATLAB prompt, type:

```matlab
mex yprimef.for yprimefg.for
```

**Special Considerations for CodeWarrior 10 Users.** There are several cases when CodeWarrior users may have to perform some additional steps to use the `mex` script.
Updating Project. While using the `mex` script with CodeWarrior on a Macintosh, you may get a warning dialog that reads, "This project was created by an older version of CodeWarrior. Do you wish to update it?" Then, do the following:

1. Click on the **Cancel** button to dismiss the dialog.
2. From the Finder, select the file `<MATLAB>:extern:src:PPCstationery.proj`.
3. Choose **Get Info** from the **File** menu.
4. Uncheck the **Stationery pad** check box in the **PPCstationery.proj Info** window.
5. Switch applications to CodeWarrior.
6. From CodeWarrior, open the `PPCstationery.proj` file using **Open** from the **File** menu.
7. When the "Do you wish to update it?" dialog appears, click **OK**.
8. Close the project by selecting **Close** from the **File** menu.
9. Switch back to the Finder.
10. Again, select the `PPCstationery.proj` file from the Finder and choose **Get Info** from the **File** menu.
11. Recheck the **Stationery pad** check box.
12. Close the **PPCstationery.proj Info** window by selecting **Close Window** from the **File** menu.

If you get the same warning dialog on a 68K Macintosh, repeat steps 2 through 12 using the file `<MATLAB>:extern:src:68Kstationery.proj`.

You will now be able to use the `mex` script without getting the warning dialog shown above.
Access Path Message. The CodeWarrior project file, ppcstationery.proj, included with MATLAB 5 was built with CodeWarrior 8. If you get a message that says

```
The following access path cannot be found
<CodeWarrior>:Metrowerks CodeWarrior:{Project Stationery}:Project Stationery Support:
```

you must edit your project settings.

1. Choose **Project Settings** from the **Edit** menu.

2. Remove the line
   `{compiler f}`:(Project Stationery):Project Stationery Support:

3. Click **OK**.

Special Considerations for MPW and LS Fortran Users. In addition to the previous steps described for Macintosh users, you must also:

- Add the folder `<MATLAB>:extern:scripts:` to the MPW `{Commands}` environment variable so that MPW will know where to find the `mex` script. To do this, edit the MPW file `UserStartup` by adding the line
  ```
  set Commands "<MATLAB>:extern:scripts:,{Commands}"; export Commands
  ```

- The MPW script, `mex`, works with both the MPW MrC/SC compiler and the Language Systems Fortran compiler. You can call the `mex` script directly from the MPW Shell environment by passing MATLAB’s `mex.m`, if desired.

- You must install the ToolServer application (included with MPW) in your MPW folder, and add the `<MATLAB>:extern:scripts:` folder to ToolServer’s `{Commands}` environment variable. To do this, edit the MPW file `UserStartupTS` by adding the line
  ```
  set Commands "<MATLAB>:extern:scripts:,{Commands}"; export Commands
  ```

For more information on installing ToolServer, see the documentation included with ToolServer.
Specifying an Options File

Some compilers require you to specify an options file. If so, at the MATLAB prompt, type:

\[ \text{mex yprime.c -f <optionsfile>} \]

and specify the name of the options file by using the \texttt{-f} switch.

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

- (Windows) You did not select a compiler during installation, or you want to use a compiler other than the one selected during installation, or you want to compile MAT or engine stand-alone programs.
- (UNIX) You are not using the system C compiler.
- (Macintosh) You did not configure a default compiler.

MATLAB includes some preconfigured options files that you can use with particular compilers. The table below lists those compilers whose files are included with MATLAB.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler</th>
<th>Options File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>Microsoft C/C++</td>
<td>msvcopts.bat</td>
</tr>
<tr>
<td></td>
<td>Microsoft Fortran</td>
<td>msfopts.bat</td>
</tr>
<tr>
<td></td>
<td>Watcom C/C++</td>
<td>watcopts.bat</td>
</tr>
<tr>
<td></td>
<td>Borland C</td>
<td>bccopts.bat</td>
</tr>
<tr>
<td></td>
<td>Watcom C for Engine and MAT Stand-alone Programs</td>
<td>watengmatopts.bat</td>
</tr>
<tr>
<td></td>
<td>Microsoft Visual C for Engine and MAT Stand-alone Programs</td>
<td>msvcengmatopts.bat</td>
</tr>
<tr>
<td></td>
<td>Borland C for Engine and MAT Stand-alone Programs</td>
<td>bccengmatopts.bat</td>
</tr>
</tbody>
</table>
Note: On Windows, if you choose a compiler at install time, the corresponding options file becomes the default, called mexopts.bat.

To use one of these options files,

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

2. Specify the -f <optionsfile> flag in the mex command using the filename of the desired options file.

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compiler</th>
<th>Options File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macintosh</td>
<td>Metrowerks CodeWarrior C</td>
<td>mexopts.cw</td>
</tr>
<tr>
<td></td>
<td>MPW MrC/SC</td>
<td>mexopts.mpwc</td>
</tr>
<tr>
<td></td>
<td>Language Systems Fortran</td>
<td>mexopts.ls</td>
</tr>
<tr>
<td>UNIX</td>
<td>System ANSI Compiler</td>
<td>mexopts.sh</td>
</tr>
<tr>
<td></td>
<td>GCC</td>
<td>gccopts.sh</td>
</tr>
<tr>
<td></td>
<td>System C++ Compiler</td>
<td>cxxopts.sh</td>
</tr>
</tbody>
</table>
Troubleshooting Your Configuration
This section focuses on some common problems that might occur when creating MEX-files.

Search Path Problem
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.

Compiler Problem
On the Sun OS 4.1.3 platform, the bundled compiler is not ANSI; you must acquire a supported compiler. The same is true on the HP-700; you must acquire a supported compiler.

General Configuration Problem
Make sure you followed the configuration steps for your platform described in this chapter. Also, refer to Chapter 6, “System Setup,” for additional information.
Creating C Language MEX-Files

2-2 C MEX-Files
2-2 The Parts of a MEX-File

2-7 Examples of C MEX-Files
2-7 A First Example
2-11 Manipulating Strings
2-13 Passing Two or More Inputs or Outputs
2-15 Manipulating Structures
2-20 Manipulating Cell Arrays
2-23 Handling Complex Data
2-25 Handling 8-, 16-, and 32-Bit Data
2-27 Manipulating Multidimensional Numerical Arrays
2-28 Handling Sparse Arrays
2-30 Calling MATLAB Functions and Other
User-Defined Functions from Within a MEX-File
2-31 Advanced Topics

2-33 How to Debug C Language MEX-Files
2-33 Debugging on UNIX
2-34 Debugging on Windows
2-35 Debugging on Macintosh
C MEX-Files

MEX-files are built by combining your C source code with a set of routines provided in the MATLAB 5 API Library.

Directory Organization

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

<table>
<thead>
<tr>
<th>Platform</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>&lt;matlab&gt;\extern</td>
</tr>
<tr>
<td>UNIX Workstation</td>
<td>$MATLAB/extern</td>
</tr>
<tr>
<td>Macintosh</td>
<td>&lt;matlab&gt;:extern</td>
</tr>
</tbody>
</table>

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

For more information about the contents of the subdirectories, refer to Chapter 6, where platform-specific information is discussed. If you are using a Macintosh platform, you may need to configure or install certain files located in these subdirectories.

The Parts of a MEX-File

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

- A computational routine that contains the code for performing the computations that you want implemented in the MEX-file.
- 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 a pointer to the right-hand input `mxArrays`, `nrhs` is the number of right-hand input `mxArrays`, `plhs` is a pointer to the left-hand output `mxArrays`, and `nlhs` is the number of left-hand output `mxArrays`. 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.

This diagram shows how inputs enter a MEX-file, what functions the gateway function performs, and how outputs return to MATLAB:
A call to MEX-file `fun`:

\[
[C, D] = \text{fun}(A, B);
\]

tells MATLAB to assign pointers of type `mxArray` to `A`, `B`, `C`, and `D`. `C` and `D` are empty.

**MATLAB**

The MEX-file returns pointers of type `mxArray` to `C` and `D`, which now contain the output data from your MEX-file.

**Inputs**

- `const mxArray *B`  
  \[
  B = \text{prhs}[1]
  \]

- `const mxArray *A`  
  \[
  A = \text{prhs}[0]
  \]

**Entry point** `mexFunction fun.c`

In the gateway function:

- Use `mxGet__` to assign pointers to the data in `prhs[0], [1], ...`.
- Call your C subroutine using the newly created pointers as function parameters.
- Use `mxCreate__` to allocate memory and set a pointer of type `mxArray` to the memory. Set `plhs[0], [1], ...` to the newly created `mxArray` structure(s).
- If not done by `mxCreate__`, load the outputs of the C subroutine in the `mxArray` structure by using `mxSet__`.

**Outputs**

- `mxArray *D`  
  \[
  D = \text{plhs}[1]
  \]

- `mxArray *C`  
  \[
  C = \text{plhs}[0]
  \]
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:

```c
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 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:

```c
nlhs = 1
nrhs = 2
plhs = (pointer_at)-->*unassigned*
prhs = (pointer_at)-->y
    (pointer_at)-->z
```

The parameter `plhs` points at nothing since the output `x` is not created until the subroutine executes. It is the responsibility of the gateway routine to
create an output MATLAB array and to assign a pointer to that MATLAB array in `plhs[0].`
Examples of C MEX-Files

The next sections of this chapter include examples of different MEX-files. The MATLAB 5 API provides a full set of routines that handle the various data types supported by MATLAB. For each data type there is a specific set of functions that you can use for data manipulation. The first example discusses the simple case of doubling a scalar. After that, the examples discuss how to pass in, manipulate, and pass back various data types, and how to handle multiple inputs and outputs. Finally, the sections discuss passing and manipulating:

- Strings
- Matrices
- Multiple input and outputs
- Structures
- Cell arrays
- Multidimensional numeric arrays
- Sparse arrays
- 8-, 16-, and 32-bit signed and unsigned integers

A First Example

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

```c
#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:

```c
#include "mex.h"
#include <math.h>

void timestwodouble y[], double x[])
{
y[0] = 2.0*x[0];
return;
}

void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    double *y;
    double *x;
    unsigned int m, n;
    /* Check for proper number of arguments. */
    if (nrhs !=1) {
        mexErrMsgTxt("Only one input argument allowed.");
    } else if (nlhs !=1) {
        mexErrMsgTxt("Only one output argument allowed.");
    }
    /* The input x must be a scalar. */
    m = mxGetM(prhs[0]);
    n = mxGetN(prhs[0]);
    if (!mxIsNumeric(prhs[0]) || mxIsComplex(prhs[0])
        || mxIsSparse(prhs[0]) || !mxIsDouble(prhs[0])
        || !(m ==1 && n ==1)) {
        mexErrMsgTxt("Input x must be a scalar.");
    }
    /* Create a matrix for the return argument. */
    plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
    /* Assign pointers to each input and output. */
    y = mxGetPr(plhs[0]);
    x = mxGetPr(prhs[0]);
    /* Call the timestwo subroutine. */
    timestwo(y, x);
}
/* End of file.*/
```
Note that this function checks for the correct number of arguments and for the correct data types. It is important to check for the correct data types because MATLAB, the language, does not type data in the same way C does. Checking for the data type avoids the problem of miscasting pointers and data types.

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, `mex.sh` on UNIX, and `mex` on the Macintosh. In all cases, 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):

```c
void timestwo_alt(double *y, double x)
{
    *y = 2.0*x;
    return;
}
void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    double *y;
    double x;
    unsigned m=1, n=1;

    /* 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 that 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);
}
/* End of file. */
```

This example passes the input scalar \(x\) by value into the `timestwo_alt` subroutine, but passes the output scalar \(y\) by reference.
Manipulating Strings

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

```c
#include "mex.h"
#include <math.h>
/* The computational routine. */
char *revord(char *buf, int buflen)
{
    int i;
    char *rev_string;
    /* Allocate memory for the reverse order string. */
    rev_string = (char *)mxCalloc(buflen+1, sizeof(char));
    /* Reverse the order of the input string. */
    for(i=0;i<buflen;i++)
        *(rev_string+i) = *(buf+buflen-i-2);
    return(rev_string);
}
```

In this example, the API function `mxCalloc` replaces `calloc`, the standard C function for dynamic memory allocation. `mxCalloc` allocates dynamic memory using MATLAB’s memory manager. You must use `mxCalloc` in any situation where C would require the use of `calloc`. `mxCalloc` automatically frees up memory upon exiting the routine. If you don’t want this to happen, use the API function `mexMakeMemoryPersistent`. 
Creating C Language MEX-Files

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

```c
/* The gateway function. */
void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    char *buf, *rev_string;
    unsigned int buflen;
    /* Find out how long the input string array is. */
    buflen = (mxGetM(prhs[0]) * mxGetN(prhs[0])) + 1;
    /* Allocate enough memory to hold the converted string. */
    buf = mxAlloc(buflen, sizeof(char));
    /* Copy the string data from prhs[0] and place it into buf. */
    status = mxGetString(prhs[0], buf, buflen);
    if (status == 0)
        mexPrintf("The converted string is %s.
", buf);
    else
        mexErrMsgTxt("Could not convert string data.");
    /* Allocate memory to hold the reverse order string. */
    rev_string = mxAlloc(buflen, sizeof(char));
    /* Call the C subroutine. Note: buf and rev_string are C
     * strings. */
    rev_string = revord(buf, buflen);
    /* Set the mxArray left-hand side pointer to the output. This
     * recasts the pointer rev_string to a type mxArray pointer. */
    plhs[0] = mxCreateString(rev_string);
}
/* End of file. */
```

Note that the gateway function creates a string `buf` of type `char` that it passes to the computational subroutine (`revord`). The subroutine `revord` returns a string of type `char *`. The API function `mxCreateString` creates a copy of the returned string (`rev_string`) and assigns a pointer of type `mxArray` to the new
pointer. Setting `plhs[0]` to the new `mxArray` pointer allows MATLAB to recognize the new string as a MATLAB variable. By not passing a variable of type `mxArray` into the computational subroutine, you can avoid having to make significant changes to your original C code. If you don't have existing C code, it is not necessary for you to write both gateway and computational routines. You can include both parts of the MEX-file in one routine, as long as it includes the proper headers and still names the entry point `mexFunction`.

**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. 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.

For example, here's a routine that multiplies an input scalar times an input matrix and outputs one matrix. That is, for this function, call

```
z = xtimesy(x,y);
```
This is the corresponding MEX-file C code:

```c
#include "mex.h"
#include <math.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++;
        }
    }
    return;
}
/* The gateway function. */
void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    int status, m, n;
    double *y, *z;
    double x;
    /* Check for proper number of arguments. */
    if (nrhs !=2) {
        mexErrMsgTxt("Two inputs required.");
    } else if (nlhs !=1) {
        mexErrMsgTxt("Only one output is allowed.");
    }
    /* Check to make sure the first input argument is a scalar. */
    if (!mxIsNumeric(prhs[0]) || mxIsComplex(prhs[0]) || mxIsSparse(prhs[0]) || !mxIsDouble(prhs[0]) || mxGetN(prhs[0])*mxGetM(prhs[0])!=1 )
        mexErrMsgTxt("Input x must be a scalar.");
    /* Get the scalar input x. Note that x is the first input. */
    x = mxGetScalar(prhs[0]);
    /* Create a pointer to a copy of the second input, matrix y. */
    y = mxGetPr(prhs[1]);
    /* Get the dimensions of the matrix input y. */
```

n = mxGetN(prhs[1]);
m = mxGetM(prhs[1]);
/* Set the output pointer to the output matrix, z. */
plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
/* Create a C pointer to a copy of the output matrix. */
z = mxGetPr(plhs[0]);
/* Call the C subroutine. */
xtimesy(x, y, z, m, n);
/* Load the new matrix data into plhs[0]. */
mxSetPr(plhs[0], z);
}
/* End of file. */

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` returned 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 create a pointer to a copy of `x`.

**Manipulating Structures**

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

Passing structures into MEX-files is just like passing any other data type, except that each field element is itself of type `mxArray`. In practice, this means that `mxGetField` returns a pointer of type `mxArray`. You can then treat this pointer like any other pointer of type `mxArray`, but if you want to pass the field value into a C routine, you must use an API function like `mxGetString` to create a pointer of type `char *` that points to a copy of the field value.

Another feature of structures in the MATLAB 5 API is that `mxCreateStructArray` allocates memory for a structure but does not set field values for individual field entries. To do that, you must use `mxSetField`. This differs from simpler constructions like character strings in that `mxCreateString` not only allocates memory for a string to be included in an
mxArray structure, but also sets the mxArray pointer contents to whatever string assignment you make. In other words, there is no need for an mxSetString since mxCreateString does that for you. In general, the more complex data structures, like cell arrays and structures, do not allocate memory and set contents to it in one step. The basic rule is that recursive data structures only allocate to one recursion level. In other words, no mxArray * variables within a data type are initialized. When in doubt, refer to the appropriate reference pages.

This example takes an input structure containing names and addresses of a client data base and returns a new structure in which the names have been arranged in alphabetical order:

```c
#include <matrix.h>
#include <string.h>
#include "mex.h"
#define FMAX 2 /* Assume there are exactly two fields. */
#define NMAX 50 /* Assume a max. of 50 names. */
void sortf(char *f1[], char *f2[], int n)
{
  int i, j;
  void swap(char **, char **);
  /* Compare and switch names if necessary. */
  for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      if(strcmp(f1[i],f1[j])<0){
        swap(&f1[i],&f1[j]);
        swap(&f2[i],&f2[j]);
      }
  return;
}
void swap(char **x, char **y)
{
  char *tmp;
  /* This subroutine swaps pointers. */
  tmp = *x;
  *x = *y;
  *y = tmp;
  return;
}
```
This is the gateway routine that calls sortf, the computational subroutine:

```c
/* The gateway routine. */
void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    int i, j, n, numfields, status, buflen, num_dims;
    const int *dims;
    const int *dims2;
    char *buf[NMAX], *buf2[NMAX], *fnames[FMAX];
    mxArray *tmp;
    char *tmp2;
    /* Check to see if input is a structure. */
    /* If not, print an error message and exit. */
    if(!mxIsStruct(prhs[0]))
        mexErrMsgTxt("Input is not a structure.");
    /* Get the number of dimensions of the structure. */
    num_dims = mxGetNumberOfDimensions(prhs[0]);
    /* Get the size of the structure, the number of */
    /* fields in each structure, and the field names. */
    dims = mxGetSize(prhs[0]);
    numfields = mxGetNumberOfElements(prhs[0]);
    for(i=0; i<numfields; i++)
        fnames[i]=mxGetFieldNameByNumber(prhs[0], i);
    /* Calculate the total number of fields */
    /* to loop through. */
    n = mxGetN(prhs[0])*mxGetM(prhs[0]);
    /* Load the buf and buf pointer arrays with the */
    /* contents of each field. If any field is not */
    /* a string, print an error message and exit. */
    for(i=0; i<n; i++)
    {
        tmp = mxGetField(prhs[0], i, fnames[0]);
        if(mxIsString(tmp))
            buflen = mxGetN(tmp)*mxGetM(tmp)+1;
        buf[i]=mxMalloc(buflen, sizeof(char));
        status=mxGetString(tmp, buf[i], buflen);
        if(status == 0)
            mexPrintf("The converted string is \n%s\n", buf[i]);
        else
            /* Print error message */
    }
}
```

Examples of C MEX-Files
mexErrMsgTxt("Could not convert name string.");
}
else
    mexErrMsgTxt("Element is not a name (string).\n
tmp = mxGetField(prhs[0], i, fnames[1]);
if(mxIsString(tmp)){
    buflen = mxGetN(tmp)*mxGetM(tmp)+1;
    buf2[i] = mxMalloc(buflen, sizeof(char));
    status = mxGetString(tmp, buf2[i], buflen);
    if(status == 0)
        mexPrintf("The converted string is \n%s\n", buf2[i]);
    else
        mexErrMsgTxt("Could not convert address string.");
}
else
    mexErrMsgTxt("Element is not an address (string).\n/*Call the C subroutine that sorts the names alphabetically.*/
sortf(buf, buf2, n);
/* Create an output structure equal in size to the input structure.*/
plhs[0] = mxCreateStructArray(num_dims, dims, numfields, fnames);
/* Load the reordered field elements into the new output structure.*/
for(i=0; i<n; i++){
    /* Load the names.*/
    tmp = mxCreateString(buf[i]);
    mxSetField(plhs[0], i, fnames[0], tmp);
    /* Load the addresses.*/
    tmp = mxCreateString(buf2[i]);
    mxSetField(plhs[0], i, fnames[1], tmp);
}
} /* End of program.*/
To see how this program works, input this structure:

```matlab
type

typing

typing y = sortst(user); at the MATLAB prompt produces this result:

```matlab
1x5 struct array with fields:
    name
    add
```

The contents of the first field of the new output structure y are:

```matlab
y.name
ans =
    Ames, Angela
ans =
    Gomez, James
ans =
    Helen, Arnold
ans =
    Simms, Edna
ans =
    Stang, Harold
```

The names are now in alphabetical order. Typing y(1); shows that the MEX-file correctly assigned the addresses to the alphabetized names:

```matlab
y(1);
ans =
    name: 'Ames, Angela'
    add: '882 Oak St. Dearborn TX'
```

The rest of the addresses can be seen by typing y.add at the MATLAB prompt.
Manipulating Cell Arrays

Cell arrays are a new data type in MATLAB 5; for a discussion of the features of cell arrays and the built-in functions MATLAB provides to manipulate them, refer to Using MATLAB. Like all other data types in MATLAB, cell arrays can be passed into and out of C MEX-files. You can access and manipulate individual cells of a cell array by using the API functions `mxGetCell` and `mxSetCell`, and you can create new cell arrays by using `mxCreateCellArray`.

This example takes an input cell array and removes trailing blanks from each cell that contains a character string:

```c
#include <matrix.h>
#include "ctype.h"
#include "string.h"
#include "mex.h"

char *rmblanks(char *buf, int buflen)
{
    int i, count=0;
    char ch, ch_old = '0';
    char *hold;
    /* Count the number of trailing blanks. */
    for(i=0; i<buflen; i++){
        ch = *(buf+i);
        if(isspace(ch) && isspace(ch_old))count++;
        if(!isspace(ch) && ch != NULL)count=0;
        ch_old = ch;
    }
    hold = mxCalloc(buflen-count-2, sizeof(char));
    strncpy(hold, buf, buflen-count-2);
    return(hold);
}
```
Below is the gateway function that calls the C computational routine rmblanks:

```c
/* The gateway routine. */
void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    int m, n, count, i, j, buflen, index, status;
    int subs[2];
    int nsubs=2;
    mxArray *cell_elt_pr, *str_pr;
    char *buf, *tmp;
    int dim=2; /* Assumes a 2-by-2 cell array. */

    /* Allocate memory for an unpopulated output cell array. */
    plhs[0] = mxCreateCellMatrix(dim, dim);

    if(!mxIsCell(prhs[0]))
        mexErrMsgTxt("Input is not a cell array.");

    m = mxGetM(prhs[0]);
    n = mxGetN(prhs[0]);
    if (m!=2 || n!=2) mexErrMsgTxt("Input is not 2-by-2.");

    /* Access each cell. */
    for(i=0;i<n;i++){
        for(j=0;j<m;j++){
            subs[0]=i;
            subs[1]=j;
            index = mxCalcSingleSubscript(prhs[0], nsubs, subs);
            cell_elt_pr = mxGetCell(prhs[0], index);

            /* Check each cell to find out whether it's a string. */
            if(mxIsChar(cell_elt_pr)){
                buflen = mxGetM(cell_elt_pr)*mxGetN(cell_elt_pr) + 1;
                buf = mxMalloc(buflen, sizeof(char));
                if (buf == NULL)
                    mexErrMsgTxt("Not enough heap space to hold string");
            }
        }
    }
}
```
/* Copy the string data into buf. */
status = mxGetString(cell_elt_pr, buf, buflen);
if(status!=0)
    mexErrMsgTxt("Could not convert string.");
mexPrintf("The converted string is\n%s.\n",buf);
/* Call the deblanker. */
tmp = rmblanks(buf, buflen);

/* Create a pointer of type mxArray to the deblanked
string. */
str_pr = mxCreateString(tmp);
/* Load the new string into the output cell array. */
mxSetCell(plhs[0], index, str_pr);
}
else
/* If the cell is not a string, directly load it into
output. */
mxSetCell(plhs[0], index, cell_elt_pr);
} /* End of inner for loop. */
} /* End of outer for loop. */
} /* End of program. */

First, load in a 2-by-2 cell array with some character strings:

x =
    'hello world'   '...goodbye !!'
[2x2 double] [    22]

Invoking rmblanks with this cell array containing two strings produces this
result:

» y = rmblanks(x)
y =
    'hello world'   '...goodbye !!'
[2x2 double] [    22]

The individual cells in cell arrays are themselves mxArray structures. This
means that API functions like mxGetCell return a pointer of type mxArray; you
can use any of the basic API functions like mxGetPr, mxGetString, etc. on the
returned pointer to gain access to the data in the individual cell.
Cell arrays are flexible data structures. By using the API functions provided in the MATLAB 5 API, you can use cell arrays much like you would use structures, but cell arrays allow a broader range of data types as cell entries. It is possible, for instance, to place an entire cell array into a cell of another cell array. Like MATLAB structures, elements of cell arrays are themselves structures of type `mxArray`.

**Handling Complex Data**

To manipulate complex data in C, you must separate it into real and imaginary parts. MATLAB’s API provides two functions, `mxGetPr` and `mxGetPi` that return pointers (of type `double *`) to the real and imaginary parts of input data. Once you have separated the data, you can manipulate each part individually and then combine them to return the data to MATLAB in the original complex format.

This example takes two complex vectors and convolves them:

```c
#include "mex.h"
#include <math.h>
define COLS 1 /* Vectors only. */
int convec( double *xr, double *xi, int mx, 
            double *yr, double *yi, int my, 
            double *zr, double *zi)
{
    int i, j;
    /* Perform the convolution of the complex vectors. */
    for(i=0; i< mx; i++)
        for(j=0; j<my; 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); }
return;
}
```
Below is the gateway function that calls this complex convolution:

```c
/* The Gateway function. */
void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    int rows, mx, my;
    /* Check for proper number of arguments. */
    if (nrhs < 2)
        mexErrMsgTxt("Two input arguments are required.");
    else if (nlhs > 2)
        mexErrMsgTxt("Only two input arguments are allowed.");
    if (mxGetN(prhs[0]) != 1 || mxGetN(prhs[1]) != 1)
        mexErrMsgTxt("Both inputs must be row vectors.");
    /* Get the length of each vector. */
    mx = mxGetM(prhs[0]);
    my = mxGetM(prhs[1]);
    /* Create pointers to the real and imaginary parts of the inputs. */
    xr = mxGetPr(prhs[0]);
    xi = mxGetPi(prhs[0]);
    yr = mxGetPr(prhs[1]);
    yi = mxGetPi(prhs[1]);
    /* Allocate enough memory to hold the convolution product. */
    rows = mx+my-1;
    zr = mxMalloc(rows, sizeof(double));
    zi = mxMalloc(rows, sizeof(double));
    /* Call the C convolution subroutine. */
    convec(xi, xr, mx, yi, yr, my, zr, zi);
    /* Assign the pointer to the output variable. */
    plhs[0] = mxCreateDoubleMatrix(rows, COLS, mxCOMPLEX);
    mxSetPr(zr);
    mxSetPi(zi);
} /* End of program. */
```
Examples of C MEX-Files

Entering these numbers at the MATLAB prompt

\[
\begin{align*}
  x &= 3.0000 - 1.0000i \\
      & 4.0000 + 2.0000i \\
      & 7.0000 - 3.0000i \\
  y &= 8.0000 - 6.0000i \\
      & 12.0000 + 16.0000i \\
      & 40.0000 - 42.0000i
\end{align*}
\]

and invoking the new MEX-file

\[
z = \text{convec}(x, y)
\]

results in

\[
\begin{align*}
  z &= 1.0e+002 \times \\
     & -0.1800 - 0.2600i \\
     & -0.9600 + 0.2800i \\
     & -1.3200 - 1.4400i \\
     & -3.7600 - 0.1200i \\
     & -1.5400 - 4.1400i
\end{align*}
\]

which agrees with the results the built-in MATLAB function \texttt{conv.m} produces.

**Handling 8-, 16-, and 32-Bit Data**

You can create and manipulate signed and unsigned 8-, 16-, and 32-bit data from within your MEX-files. The MATLAB 5 API provides a set of functions that support these data types. The API function \texttt{mxCreateNumericArray} constructs an unpopulated N-dimensional numeric array with a specified data size. Refer to the entry for \texttt{mxClassID} in the online reference pages for a discussion of how the MATLAB 5 API represents these data types.

Once you have created an unpopulated MATLAB array of a specified data type, you can access the data using \texttt{mxGetPr} and \texttt{mxGetPi}. These two functions create copies of the real and imaginary data and set pointers to the copies of the data. You can perform arithmetic on data of 8-, 16- or 32-bit precision in MEX-files and return the result to MATLAB, which will recognize the correct data class. Although from within MATLAB it is not currently possible to perform arithmetic or to call MATLAB functions that perform data manipulation,
on data of 8-, 16-, or 32-bit precision, you can display the data at the MATLAB prompt.

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

```c
#include "mex.h"
#include <matrix.h>
#define ndim 2
#define dims {2 2}

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

    for(i=0;i<dims(0);i++)
        for(j=0;j<dims(1);j++)
            *(x+i+j) = scalar* *(x+i+j);
}

/* The gateway function. */
void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    unsigned short data[]={1 2 3 4};

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

    /* Create a 2-by-2 array of unsigned 16-bit integers. */
    array_ptr = mxCreateNumericArray(ndim, dims, mxUINT16_CLASS,
        mxREAL);
    if (array_ptr == NULL)
        mexErrMsgTxt("Could not create mxArray.
    
    /* Populate the real part of the created array. */
    start_of_pr = (unsigned char *)mxGetPr(array_ptr);
    bytes_to_copy = TOTAL_ELEMENTS * mxGetElementSize(array_ptr);
```
Examples of C MEX-Files

Examples of C MEX-Files

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

Manipulating Multidimensional Numerical Arrays

Multidimensional numerical arrays are a new data type in MATLAB 5. For a discussion of the features of multidimensional numerical arrays and the built-in functions MATLAB provides to manipulate them, refer to Using MATLAB. Like all other data types in MATLAB, arrays can be passed into and out of MEX-files written in C. You can manipulate multidimensional numerical arrays by using `mxGetPr` and `mxGetPi` to create copies of the real and imaginary parts of the data stored in the original multidimensional array, and you can create a new multidimensional array from within your MEX-file by using `mxCreateNumericArray`.

This example takes a real 2-by-3-by-2 dimensional array with unsigned 8-bit elements and squares each element (header files are omitted for brevity):

```c
#define FIRST_DIM 2
#define SECOND_DIM 3
#define THIRD_DIM 2
#define TOTAL_ELEMENTS (FIRST_DIM * SECOND_DIM * THIRD_DIM)
void sqmat(char *x, int l, int m, int n, char *y)
{
    int i, j, k;
    for (i=0; i<FIRST_DIM; i++)
        for(j=0; j<SECOND_DIM; j++)
            for(k=0; k<THIRD_DIM; k++)
                *(y+i+j+k) = *(x+i+j+k) * *(x+i+j+k);
    return;
}
/* The gateway routine. */
void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    memcpy(start_of_pr, real_data, bytes_to_copy);
    /* Set the output pointer to the loaded array. */
    plhs[0] = array_ptr;
}
```

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

Manipulating Multidimensional Numerical Arrays

Multidimensional numerical arrays are a new data type in MATLAB 5. For a discussion of the features of multidimensional numerical arrays and the built-in functions MATLAB provides to manipulate them, refer to Using MATLAB. Like all other data types in MATLAB, arrays can be passed into and out of MEX-files written in C. You can manipulate multidimensional numerical arrays by using `mxGetPr` and `mxGetPi` to create copies of the real and imaginary parts of the data stored in the original multidimensional array, and you can create a new multidimensional array from within your MEX-file by using `mxCreateNumericArray`.

This example takes a real 2-by-3-by-2 dimensional array with unsigned 8-bit elements and squares each element (header files are omitted for brevity):

```c
#define FIRST_DIM 2
#define SECOND_DIM 3
#define THIRD_DIM 2
#define TOTAL_ELEMENTS (FIRST_DIM * SECOND_DIM * THIRD_DIM)
void sqmat(char *x, int l, int m, int n, char *y)
{
    int i, j, k;
    for (i=0; i<FIRST_DIM; i++)
        for(j=0; j<SECOND_DIM; j++)
            for(k=0; k<THIRD_DIM; k++)
                *(y+i+j+k) = *(x+i+j+k) * *(x+i+j+k);
    return;
}
/* The gateway routine. */
void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    memcpy(start_of_pr, real_data, bytes_to_copy);
    /* Set the output pointer to the loaded array. */
    plhs[0] = array_ptr;
}
```
int ndim = 3, dims[3] = {FIRST_DIM, SECOND_DIM, THIRD_DIM};
unsigned char data[] = {9, 7, 5, 2, 6, 3, 4, 8, 2, 1, 10, 5};
unsigned char *start_of_pr;
unsigned char *start_of_pry;
mxArray *array_ptr;
size_t bytes_to_copy;

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

/* Create a 2-by-3-by-2 array of unsigned 8-bit integers. */
ary_ptr = mxCreateNumericArray(ndim, dims,
mUINT8_CLASS, mxREAL);
if (array_ptr == NULL)
mexErrMsgTxt("Could not create mxArray.
");

/* Populate the real part of the created array. */
start_of_pr = (unsigned char *)mxGetPr(array_ptr);
bytes_to_copy = TOTAL_ELEMENTS * mxGetElementSize(array_ptr);
memcpy(start_of_pr, data, bytes_to_copy);

/* Set the output pointer to the loaded array. */
plhs[0] = ary_ptr;

Handling Sparse Arrays
The MATLAB 5 API provides a set of functions that allow you to create and manipulate sparse arrays from within your MEX-files. These API routines access and manipulate \texttt{ir} and \texttt{jc}, two of the parameters associated with sparse arrays. For more information on how MATLAB stores sparse arrays, refer to “The MATLAB Array” section in Chapter 1 of this guide.

This example creates a 2-by-4 sparse array and sums its nonzero entries:

```c
#include "mex.h"
#include <math.h>
#define NZMAX 4
#define ROWS 4
#define COLS 2

/* The computational subroutine. */
```
double sumit(x)
{
    double *x;
    double sum=0.0;

    for(i=0; i<NZMAX; i++) sum+= *(x+i);
    return(sum)
}

/* The gateway function. */
void mexFunction(int nlhs, mxArray plhs[],
    int prhs, const mxArray prhs[])
{
    int            rows=ROWS, cols=COLS;
    mxArray       *ptr_array; /* Pointer to created sparse array. */
    static double  static_pr_data[NZMAX] = {5.8, 6.2, 5.9, 6.1};
    static int     static_ir_data[NZMAX] = {0, 2, 1, 3};
    static int     static_jc_data[COLS+1] = {0, 2, 4};
    double        *start_of_pr;
    int           *start_of_ir, *start_of_jc;
    mxArray       *array_ptr;
    double        sum;

    /* Create a sparse array and name it "Sparrow". */
    array_ptr = mxCreateSparse(rows, cols, NZMAX, mxREAL);
    mxSetName(array_ptr, "Sparrow");

    /* Place pr data into the newly created sparse array. */
    start_of_pr = (double *)mxGetPr(array_ptr);
    memcpy(start_of_pr, static_pr_data, NZMAX*sizeof(double));

    /* Place ir data into the newly created sparse array. */
    start_of_ir = (int *)mxGetIr(array_ptr);
    memcpy(start_of_ir, static_ir_data, NZMAX*sizeof(int));

    /* Place jc data into the newly created sparse array. */
    start_of_jc = (int *)mxGetJc(array_ptr);
    memcpy(start_of_jc, static_jc_data, NZMAX*sizeof(int));

    /* Call the computational subroutine. */
    sum = sumit(start_of_pr)
    /* When finished with the mxArray, deallocate it. */
Calling MATLAB Functions and Other User-Defined Functions from Within a MEX-File

It is possible to call MATLAB functions, operators, M-files, and even other MEX-files from within your C source code by using the API function mexCallMATLAB. This example creates a populated array, calls mexCallMATLAB to display its contents, calls mexCallMATLAB to calculate the eigenvalues and eigenvectors of the matrix, and finally calls mexCallMATLAB a third time to display the eigenvalues:

```c
#include <mex.h>
#include <math.h>
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    double pr[] = {5.2, 7.9, 1.3, 4.2};
    double pi[] = {3.4, 6.5, 2.2, 9.1};
    mxArray *array_ptr;
    int num_out, num_in;
    mxArray *output_array[2], *input_array[2];

    /* Create a 2-by-2 populated matrix. */
    mxArray_ptr = mxCreateDoubleMatrix(2, 2, mxCOMPLEX);
    memcpy(mxGetPr(array_ptr), pr, 4);
    memcpy(mxGetPi(array_ptr), pi, 4);

    /* Equivalent to disp(array). */
    num_out = 0;
    num_in = 1;
    mxArray_ptr[0] = array_ptr;
    mxArray_ptr[1] = mxArray_ptr;
    mxArray_ptr[2] = mxArray_ptr;
    mxArray_ptr[3] = mxArray_ptr;

    mexCallMATLAB(num_out, output_array, num_in, input_array, 
                  "disp");
```
/* Equivalent to \([v, d] = \text{eig}(\text{array})\). */
num_out = 2;
num_in = 1;
input_array[0] = array_ptr;
mexCallMATLAB(num_out, output_array, num_in, input_array, 
"eig");

/* Equivalent to \(\text{disp}(v)\). */
num_out = 0;
num_in = 1;
input_array[0] = output_array[0];
mexCallMATLAB(num_out, output_array, num_in, input_array, 
"disp");
} /* End of program. */

Unlike most of the other examples, this MEX-file does not invoke a computational C subroutine. This was a stylistic choice; it is possible to use mexCallMATLAB (or any other API function) from within your computational C subroutine. Note that you can only call most MATLAB functions with double-precision data. M-functions like eig will not work correctly with data that is not double-precision.

Advanced Topics
This sections covers advanced features of MEX-files that you can use when your applications require sophisticated MEX-files.

Help Files
Because the MATLAB interpreter chooses the MEX-file when both an M-file and a MEX-file having the same name are encountered in the same directory, it is possible to use M-files for documenting the behavior of your MEX-files. The MATLAB help command will automatically find and display the appropriate M-file when help is requested and the interpreter will find and execute the corresponding MEX-file when the function is actually invoked.
Linking Multiple Files

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

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

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

You may find it useful to use a software development tool like `MAKE` to manage MEX-file projects involving multiple source files. Simply create a `MAKEFILE` that contains a rule for producing object files from each of your source files and then invokes `mex` to combine your object files into a MEX-file. This way you can ensure that your source files are recompiled only when necessary.

Variable Scope

Being inside a MEX-file is much like being inside an M-file function. However, there are two important differences regarding the scope of variables between the behavior of M-files and MEX-files:

- MEX-files can have static (persistent) variables; M-files cannot.
- MEX-files can modify variables in the caller's workspace; M-files cannot.
  MEX-files can use the routines `mexPutArray` and `mexEvalString` if they need to modify the caller's workspace.
How to Debug C Language MEX-Files

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

Note: The sections, “Troubleshooting MEX-File Creation” and “Understanding MEX-File Problems,” in Chapter 6 provide additional information on isolating problems with MEX-files.

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

```bash
mex -g filename.c
```

Debugging on UNIX

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

```bash
matlab -Ddbx
```

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

```bash
dbmex on
```

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

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

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

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

```
  dbmex stop
```

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

## Debugging on Windows

**Watcom Compiler.** If you are using the Watcom compiler:

1. Start MATLAB in the debugger by typing on the command line

   `WDW MATLAB.EXE`

2. From the Break menu, select On Image Load and type the name of your MEX-file DLL.

3. Click GO. The debugger will stop as it loads your MEX-file.

**Microsoft Compiler.** If you are using the Microsoft compiler:

1. Start MATLAB in the debugger by typing on the command line

   `MSDEV -debug MATLAB.EXE`

2. Open your MEX-file source code.

3. Set a break point on the desired line of code by right-clicking your mouse on the line of code.

4. From the Build menu, select Debug option, and click GO.
Debugging on Macintosh

CodeWarrior Compiler. If you are using the CodeWarrior compiler on Power Macintosh or 68K Macintosh systems:

1. Build the MEX-file from MATLAB using `mex.m`. In addition to the normal switches you use to build the MEX-file, specify the `-g` switch.

2. Start the Metrowerks Debugger application. When it asks you to select a SYM file, select the `.xSYM` file created by CodeWarrior when you built the MEX-file (e.g., `timestwo.mex` would have a SYM file called `timestwo.mex.xSYM`).

3. Set breakpoint(s) within your MEX-file's source code.

4. Run the MEX-function from within MATLAB.

The Metrowerks Debugger will automatically pop to the foreground when a breakpoint is reached.
Jasik Debugger on 68K. If you are using the Jasik Debugger on MEX-files created with MPW SC, LS Fortran, or CodeWarrior C on 68K Macintosh systems:

1. Create a .dsi file called MATLAB.dsi with any text editor (such as MPW or SimpleText). It should look similar to the following example, and contain at least the following information:

   MATLAB.dsi
   
   =G
   MEXO; Debug code resources of type 'MEXO' (M-E-X-zero)
   
   =F
   Dbg_Rsrcs = 1; Enable resource debugging
   
   =E
   ; End .dsi file

2. Place the MATLAB.dsi file in the same folder as the MATLAB binary.

3. You will need a dummy .MAP file so that the debugger will open and execute the MATLAB.dsi file. Place the following in a text file called MATLAB.MAP and put this file into the same folder as the MATLAB binary:

   MATLAB.MAP
   
   Segment 'Main' size=6000000 rsrcid=0 JTindex=0 #TEnts=00000
   _Startup_
   $000000 size=$000000 extern
4 Build the MEX-file from MATLAB (using `mex.m`) or MPW (using `mex`). In addition to the normal switches you use to build the MEX-file, specify the `-g` switch.

5 Rename the resulting .SYM file from `<filename>.SYM` to `<filename>/MEX0_0.SYM`. For example, `conv.mex.SYM` becomes `conv.mex/MEX0_0.SYM` (MEX0_0 is M-E-X-zero-underscore-zero). Make sure this file is in the same folder as the MEX-file.

6 Create a `.dsi` file for the MEX-file itself. It should look similar to the following example, and contain at least the following information:

   `<filename>/MEX0_0.dsi`

   ```
   =bkpt
   mexFunction ; set breakpoint at function mexFunction
   =S
   MyDisk:myFirstSourceFolder: ; set list of folders where source files reside
   MyDisk:mySecondSourceFolder:
   =E
   ; End .dsi file
   ```

7 Name this file `<filename>/MEX0_0.dsi`. Using the `conv` example above, this becomes `conv.mex/MEX0_0.dsi`. Make sure this file is in the same folder as the MEX-file.

8 Run the MEX-function from within MATLAB. The debugger will automatically take control when the MEX-function is called, executing its `.dsi` file and stopping at the breakpoint for `mexFunction`. 
Jasik Debugger on Power Macintosh. If you are using the Jasik Debugger on MEX-files created with MPW MrC, LS Fortran, or CodeWarrior C on Power Macintosh systems:

1. From within the debugger, make sure the `Dbg_SLM` option in the `-Dbgr Status` window is set to ON.

2. Build the MEX-file from MATLAB (using `mex.m`) or MPW (using `mex`). In addition to the normal switches you use to build the MEX-file, specify the `-g` switch.

3. Rename the resulting `.xSYM` file to `mexFunction.xSYM`. Make sure this file is in the same folder as the MEX-file.

4. Create a `.dsi` file for the MEX-file itself. It should look similar to the following example, and contain at least the following information:

   ```
   !-bkpt
   mexFunction; set breakpoint at function mexFunction
   
   =S
   MyDisk:myFirstSourceFolder: ; set list of folders where source
   ; files reside
   MyDisk:mySecondSourceFolder:
   
   =E
   ; End .dsi file
   ```

5. Name this file `mexFunction.dsi`, and make sure it is in the same folder as the MEX-file.

6. Run the MEX-function from within MATLAB. The debugger will automatically take control when the MEX-function is called, executing its `.dsi` file and stopping at the breakpoint for `mexFunction`.
Creating Fortran MEX-Files

3-2 Fortran MEX-Files
3-2 MEX-Files and Data Types
3-3 The Components of a Fortran MEX-File

3-9 Examples of Fortran MEX-Files
3-9 A First Example—Passing a Scalar
3-11 Passing Strings
3-13 Passing Arrays of Strings
3-14 Passing Matrices
3-17 Passing Two or More Inputs or Outputs
3-20 Handling Complex Data
3-22 Dynamic Allocation of Memory
3-26 Handling Sparse Matrices
3-28 Calling MATLAB Functions from Fortran MEX-Files
3-32 Advanced Topics

3-33 How to Debug Fortran Language MEX-Files
3-33 Debugging on UNIX
3-34 Debugging on Windows
3-35 Debugging on Macintosh
Fortran MEX-Files

This section discusses the procedure for creating Fortran MEX-files. MEX-files are built by combining your Fortran source code with a set of routines provided in the MATLAB 5 API Library.

Directory Organization
The table below lists the location of the files on your disk that are associated with the creation of Fortran language MEX-files:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>\matlab|extern</td>
</tr>
<tr>
<td>UNIX Workstation</td>
<td>$MATLAB|extern</td>
</tr>
<tr>
<td>Macintosh</td>
<td>\matlab:extern</td>
</tr>
</tbody>
</table>

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

For more information about the contents of the subdirectories, refer to Chapter 6, which provides platform-specific information. If you are using a Macintosh platform, you may need to configure or install certain files located in these subdirectories.

MEX-Files and Data Types
MEX-files in Fortran can accept only double-precision data and strings (unlike their C counterparts, which can accept any data type supported by MATLAB). You can treat Fortran MEX-files, once compiled, exactly like M-functions.
The Components of a Fortran MEX-File

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

- A computational routine that contains the code for performing the computations that you want implemented in the MEX-file.
- 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 a pointer to the right-hand input `mxArrays`, `nrhs` is the number of right-hand input `mxArrays`, `plhs` is a pointer to the left-hand output `mxArrays`, and `nlhs` is the number of left-hand output `mxArrays`. The gateway calls the computational routine as a subroutine.

The computational and gateway routines may be separate or combined. This diagram shows how inputs enter an API function, what functions the gateway function performs, and how output returns to MATLAB.
A call to MEX-file `fun`:

```matlab
[C,D]=fun(A,B);
```

MATLAB sends `A` and `B` to the MEX-file.

The MEX-file returns `C` and `D`. MATLAB can now use `C` and `D` like any other workspace variable.

MATLAB sends `A` and `B` to the MEX-file.
The Pointer Concept

The MATLAB API works with a unique data type, the `mxArray`. Because there is no way to create a new data type in Fortran 77, MATLAB passes a special identifier, called a pointer, to a Fortran program. You can get information about an `mxArray` by passing this pointer to various API functions (called “Access Routines”). These access routines allow you to get a native Fortran data type containing exactly the information you want—the size of the `mxArray`, whether or not it is a string, or its data contents.

There are several implications when using pointers in Fortran:

- **The %val construct.**
  
  The `%val` construct is an extension to standard Fortran 77 and Fortran 90. If your Fortran compiler supports the `%val` construct, then there is one type of pointer you can use without requiring an access routine, namely a pointer to data (i.e., the pointer returned by `mxGetPr` or `mxGetPi`). You can use `%val` to pass this pointer’s contents to a subroutine, where it is declared as a Fortran double-precision matrix.

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

- **Variable declarations.**

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

  On UNIX, if you want to run your Fortran program on machines with differing pointer sizes, and if your Fortran compiler does not have its own preprocessor, then you may use a C preprocessor to map pointers to the appropriate declaration. See the examples ending with `.F` in the examples directory for a possible approach.

**Note:** Declaring a pointer to be the incorrect size can cause your program to crash.
The Gateway Routine

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

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

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

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

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

```matlab
[a, b, c, ...] = fun(d, e, f, ...)
```

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

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

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

the MATLAB interpreter calls `mexFunction` with the arguments:

```fortran
nlhs = 1
nrhs = 2
plhs = (pointer_at) => *unassigned*
prhs = (pointer_at) => y
    (pointer_at) => 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 (using `mxCreateFull`) and to set a pointer to that array in `plhs(1)`. Note that it is possible to return an output value even if `nlhs = 0`. 
This corresponds to returning the result in the ans variable. In this case, MATLAB prints a warning message stating that no output has been assigned.

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

The subroutines and functions in the MEX-file interface library provide a set of access methods (subroutines) for manipulating MATLAB arrays. These subroutines, which are fully documented in the online reference pages, always start with the letters mx. This is shorthand for mxArray and it means that the subroutine enables you to access and/or manipulate some of the information in the MATLAB array. For example, mxGetPi retrieves any imaginary data inside the MATLAB array. Additional routines are provided for transferring data between MATLAB arrays and Fortran arrays.

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

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

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

When a MEX-file completes its task, it should return control to MATLAB. Data structures that the MEX-file creates and memory allocated with routines from the MEX-file interface library are cleaned up (freed) automatically.

The %val Construct
The %val construct is supported by most, but not all, Fortran compilers. %val causes the value of the variable, rather than the address of the variable, to be passed to the subroutine. If you are using a Fortran compiler that does not support the %val construct, you must copy the array values into a temporary...
true Fortran array using special routines. For example, consider a gateway routine that calls its computational routine, \texttt{yprime}, by
\begin{verbatim}
call yprime(%val(yp), %val(t), %val(y))
\end{verbatim}
If your Fortran compiler does not support the \texttt{%val} construct, you would replace the call to the computational subroutine with
\begin{verbatim}
C Copy array pointers to local arrays.
call mxCopyPtrToReal8(t, tr, 1)
call mxCopyPtrToReal8(y, yr, 4)
C
C Call the computational subroutine.
call yprime(ypr, tr, yr)
C
C Copy local array to output array pointer.
call mxCopyReal8ToPtr(ypr, yp, 4)
\end{verbatim}
You must also add the following declaration line to the top of the gateway routine
\begin{verbatim}
real*8 ypr(4), tr, yr(4)
\end{verbatim}
Note that if you use \texttt{mxCopyPtrToReal8} or any of the other \texttt{mxCopy__} routines, the size of the arrays declared in the Fortran gateway routine must be greater than or equal to the size of the inputs to the MEX-file coming in from MATLAB. Otherwise \texttt{mxCopyPtrToReal8} will not work correctly.
Examples of Fortran MEX-Files

The next sections of this chapter include examples of different MEX-files. The MATLAB 5 API provides a set of routines for Fortran that handle double-precision data and strings in MATLAB. For each data type, there is a specific set of functions that you can use for data manipulation.

A First Example—Passing a Scalar

Let’s look at a simple example of Fortran code and its MEX-file equivalent. Here is a Fortran computational routine that takes a scalar and doubles it:

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

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

```c
subroutine timestwo(y, x)
  real*8 x, y
  integer m, n
  y = 2.0*x
  return
end
```

```c
C subroutine mexFunction(nlhs, plhs, nrhs, prhs)
  integer plhs(*), prhs(*)
  integer nlhs, nrhs
  integer mxGetM, mxGetN, mxGetPr, mxCreateFull, mxIsNumeric
  real*8 x, y

  C     Check for proper number of arguments.
  if(nrhs .ne. 1) then
    call mexErrMsgTxt('Only one input argument allowed.')
  elseif(nlhs .ne. 1) then
    call mexErrMsgTxt('Only one output argument allowed.')
  endif
```
C Get the size of the input matrix.
  m = mxGetM(prhs(1))
  n = mxGetN(prhs(1))
  size = m*n

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

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

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

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

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

mex timestwo.f

This carries out the necessary steps to create the MEX-file called timestwo
with an extension corresponding to the machine type on which you're running.
You can now call timestwo as if it were an M-function:

x = 2;
y = timestwo(x)
y =
  4
Passing Strings
Passing strings from MATLAB to a Fortran MEX-file is straightforward. This program accepts a string and returns the characters in reverse order:

```fortran
subroutine revord(x, strlen, y)
    character x(100), y(100)
    integer strlen
    
    do 10 i=1, strlen
        y(i) = x(strlen-i+1)
    10 continue
    return
end
```

Below is the gateway function that calls the computational routine:

```fortran
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
    integer plhs(*), prhs(*)
    integer nlhs, nrhs, m, n, status
    integer mxGetM, mxGetN, mxIsString
    integer mxIsString, mxGetString, mxCreateString
    character x(100), y(100)
    
    C Check for proper number of arguments.
    if (nrhs .ne. 1) then
        call mexErrMsgTxt('Only one input argument allowed.
    elseif (nlhs .ne. 1) then
        call mexErrMsgTxt('Only one output argument allowed.
    endif

    C The input x must be a string.
    if (mxIsString(prhs(0) .ne. 1)) then
        call mexErrMsgTxt('Input is not a string.')
    endif

    C In this program, only vector strings are allowed.
    C Check that the string has either one row or one column.
    m = mxGetM(prhs(0))
    n = mxGetN(prhs(0))
    if (m .ne. 1 .and. n .ne. 1) then
        call mexErrMsgTxt('Input must be a vector string.')
    endif
```
After checking for the correct number of inputs, this MEX-file gateway function verifies that the input was either a row or column vector string. It then finds the size of the string and places the string into a Fortran character array. Note that in the case of character strings, it is not necessary to copy the data into a Fortran character array by using `mxCopyPtrToCharacter`. In fact, `mxCopyPtrToCharacter` works only with MAT-files. (For more information about MAT-files, see Chapter 4, “Data Export and Import”.)

For an input string

\[ x = 'hello world'; \]

typing

\[ y = \text{revord}(x) \]

produces

\[ y = \text{dlrow olleh} \]
Passing Arrays of Strings

Passing arrays of strings involves a slight complication from the previous example in the "Passing Strings" section of this chapter. Because MATLAB stores elements of a matrix by column instead of by row, it is essential that the size of the string array be correctly defined in the Fortran MEX-file. The key point is that the row and column sizes as defined in MATLAB must be reversed in the Fortran MEX-file; consequently, when returning to MATLAB, the output matrix must be transposed to the dimensions MATLAB recognizes.

This example, putchrmx.f, places a string array/character matrix into MATLAB as output arguments rather than placing it directly into the workspace. Inside MATLAB, call this function by typing:

```
putchrmx;
```

You will get the matrix `mystring` of size 5-by-15. There are some manipulations that need to be done here. The original string matrix is of the size 5-by-15. Because of the way MATLAB reads and orients elements in matrices, the size of the matrix must be defined as $M=15$ and $N=5$ from the MEX-file. After the matrix is put into MATLAB, the matrix must be transposed.

```fortran
subroutine mexFunction(nlhs, prhs, nrhs, prhs)
  integer nlhs, nrhs
  integer plhs(*), prhs(*)
  integer ii, stat
  integer p_strm
  character thestring*75
  character string(5)*15

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

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

  ~

  ~
```

3-13
Creating Fortran MEX-Files

C Create the string matrix to be passed into MATLAB and set
C the size to be M=15 and N=5.
p_strm = mxcreatestring(thestring)
stat = mxsetm(p_strm, 15)
stat = mxsetn(p_strm, 5)

C Set the name of the string matrix inside MATLAB.
stat = mxsetname(p_strm, 'mystring')

C Pass the matrix into MATLAB.
stat = mexputmatrix(p_strm)

C Transpose the resulting matrix in MATLAB.
call mexevalstring('mystring=mystring'';')
end

Note that this MEX-file does not call a computational subroutine. Because the
MEX-file creates the string it wants to pass to MATLAB, a computational
subroutine is not needed. Typing

putchrmx;

at the MATLAB prompt produces this result:

MATLAB
The Scientific
Computing
Environment
by TMW, Inc.

Passing Matrices

In MATLAB, you can pass matrices into and out of MEX-files written in
Fortran. You can manipulate the MATLAB arrays by using mxGetPr and
mxGetPi to assign pointers to the real and imaginary parts of the data stored
in the MATLAB arrays, and you can create new MATLAB arrays from within
your MEX-file by using mxCreateFull.
This example takes a real 2-by-3 matrix and squares each element:

```fortran
subroutine matsq(y, x, m, n)
    real*8 x(2,3), y(2,3)
    integer m, n
    do 20 i=1,m
        do 10 j=1,n
            y(i,j) = x(i,j)**2
        10   continue
    20   continue
    return
end
```
Creating Fortran MEX-Files

This is the gateway routine that calls the computational subroutine:

```fortran
C The gateway subroutine.
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer plhs(*), prhs(*)
integer nlhs, nrhs
integer m,n, size, xp, yp
integer mxGetM, mxGetN, mxCreateFull, mxIsNumeric
real*8 x(3,2), y(3,2)
C Check for proper number of arguments.
if(nrhs .ne. 1) then
    call mexErrMsgTxt('Only one input argument allowed.')
elseif(nlhs .ne. 1) then
    call mexErrMsgTxt('Only one output argument allowed.')
endif
C Check to ensure the matrix is numeric (not strings).
if(mxIsNumeric(prhs(1)) .eq. 0) then
    call mexErrMsgTxt('Input must be a numeric matrix.')
endif
C Get size of input matrix.
m = mxGetM(prhs(1))
n = mxGetN(prhs(1))
size = m*n
C Create MATLAB array for return argument and get input data.
plhs(1) = mxCreateFull(m, n, 0)
yp = mxGetPr(plhs(1))
xp = mxGetPr(prhs(1))
call mxCopyPtrToReal8(xp, x, size)
C Call computational subroutine.
call matsq(y, x, m, n)
C Load the data into yp, which is the output to MATLAB.
call mxCopyReal8BToPtr(y, yp, size)
return
end
```
After performing error checking to ensure that the correct number of inputs and outputs was assigned to the gateway subroutine and to verify the input was in fact a numeric matrix, `matsq.f` creates a matrix for the argument returned from the computational subroutine. You must assign a pointer (in this example, `yp`) to the output even though the output matrix is at this point filled with zeros. The input matrix data is then copied to a Fortran matrix by using `mxCopyPtrToReal8`. Now the computational subroutine can be called, and the return argument can then be placed into `yp`, the pointer to the output, using `mxCopyReal8ToPtr`.

For a 2-by-3 real matrix,

\[ x = \begin{bmatrix} 1 & 2 & 3; & 4 & 5 & 6 \end{bmatrix}; \]

typing

\[ y = \text{matsq}(x) \]

produces this result:

\[ y = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix} \]

**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. Accordingly, `plhs(1)` contains a pointer to the first left-hand side argument, `plhs(2)` contains a pointer to the second left-hand side argument, and so on. Likewise, `prhs(1)` contains a pointer to the first right-hand side argument, `prhs(2)` points to the second, and so on.

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

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

3-17
Below is the gateway routine that calls `timesy`, the computation subroutine that multiplies a scalar times a matrix:

```c
C The gateway routine.
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
    integer plhs(*), prhs(*)
    integer nlhs, nrhs
    integer m, n, size
    integer mxGetM, mxGetN, mxIsNumeric
    integer mxCreateFull
    real*8 x, y(3,3), z(3,3)
    integer xp, yp, zp

    C Check for proper number of arguments.
    if (nrhs .ne. 2) then
        call mexErrMsgTxt('Only two input arguments allowed.')
    elseif (nlhs .ne. 1) then
        call mexErrMsgTxt('Only one output argument allowed.')
    endif

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

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

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

C Create matrix for the return argument.
plhs(1) = mxCreateFull(m, n, 0)
zp = mxGetPr(plhs(1))
xp = mxGetPr(prhs(1))
yp = mxGetPr(prhs(2))

C Load the data into Fortran matrices.
call mxCopyPtrToReal8(xp, x, 1)
call mxCopyPtrToReal8(yp, y, size)

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

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

As this example shows, creating MEX-file gateways that handle multiple inputs and outputs is straightforward. All you need to do is keep track of which indices of the vectors prhs and plhs correspond to which input and output arguments of your function. In this example, the input variable x corresponds to prhs(1) and the input variable y to prhs(2).

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

\[ x = \text{ones}(3); \ y = 3; \]

typing
\[ z = \text{xtimesy}(x, y) \]
yields this result:

\[
\begin{bmatrix}
3 & 0 & 0 \\
0 & 3 & 0 \\
0 & 0 & 3
\end{bmatrix}
\]
Handling Complex Data

Unlike C, Fortran has built-in complex data handling, so you don’t have to separate complex data into real and imaginary parts. MATLAB’s API provides two functions, `mxGetPr` and `mxGetPi`, which assign pointers to the real and imaginary parts of input data. Using the `mxCopyPtrToComplex16` routine, you can load the complex data into a single complex matrix using the pointers to the real and imaginary parts of the data.

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

```fortran
subroutine convec(xx, yy, zz, s1, s2)
complex*16 xx(3), yy(3), zz(5)
integer s1, s2, index
C
zz(1) = (0.0,0.0)
do 20 i=0,s1
   do 10 j=0,s2
      index=i+j+1
      zz(index) = zz(index)+xx(i+1)*yy(j+1)
10 continue
20 continue
return
end

C The gateway routine.
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer plhs(*), prhs(*)
integer nlhs, nrhs
integer m1, n1, m2, n2, sizex, sizey, sizez
integer mxGetM, mxGetN, mxIsComplex
integer mxCreateFull
complex*16 x(3), y(3), z(5)
integer xpr, xpi, ypr, ypi, zpr, zpi
C
C Check for proper number of arguments.
if (nrhs .ne. 2) then
   call mexErrMsgTxt('Only two input arguments allowed."
elseif (nlhs .ne. 1) then
   call mexErrMsgTxt('Only one output argument allowed."
endif
```
Check to see both inputs are complex.
if (mxIsComplex(prhs(1)) .ne. 1) then
    call mexErrMsgTxt('Input #1 is not complex.')
elseif (mxIsComplex(prhs(2)) .ne. 1) then
    call mexErrMsgTxt('Input #2 is not complex.')
endif

C Check that inputs are both vectors.
m1 = mxGetM(prhs(1))
n1 = mxGetN(prhs(1))
m2 = mxGetM(prhs(2))
n2 = mxGetN(prhs(2))
if (m1 .ne. 1 .and. n1 .ne. 1) then
    call mexErrMsgTxt('Input #1 is not a vector.')
elseif (m2 .ne. 1 .and. n2 .ne. 1) then
    call mexErrMsgTxt('Input #2 is not a vector.')
endif

C Create matrix for the return argument.
sizex = m1*n1
sizey = m2*n2

C Load the data into Fortran matrices.
call mxCopyPtrToComplex16(xpr, xpi, x, sizex)
call mxCopyPtrToComplex16(ypr,ypi, y, sizey)

C Call the computational subroutine.
call convec(x, y, z, sizex-1, sizey-1)

C Load the output into a MATLAB array.
call mxCopyComplex16ToPtr(z, zpr, zpi, sizez)
return
end

Entering these numbers at the MATLAB prompt

\[
x = \\
3.0000 - 1.0000i \\
4.0000 + 2.0000i \\
7.0000 - 3.0000i \\
y = \\
8.0000 - 6.0000i \\
12.0000 + 16.0000i \\
40.0000 - 42.0000i
\]

and invoking the new MEX-file

\[ z = \text{convec}(x, y) \]

results in

\[
z = \\
1.0000e+002 \times \\
-0.1800 - 0.2600i \\
-0.9600 + 0.2800i \\
-1.3200 - 1.4400i \\
-3.7600 - 0.1200i \\
-1.5400 - 4.1400i
\]

which agrees with the results the built-in MATLAB function \texttt{conv.m} produces.

**Dynamic Allocation of Memory**

It is possible to allocate memory dynamically in a Fortran MEX-file, but you must use \%\texttt{val} to do it. The \%\texttt{val} construction is not available on all compilers; refer to the documentation that accompanies your compiler to see if it is supported.
This example takes an input matrix of real data and doubles each of its elements:

```fortran
subroutine mexfunction(nlhs, plhs, nrhs, prhs)
    integer nlhs, nrhs
    integer plhs(*), prhs(*)
    integer m_in, n_in
    integer pr_in, pr_out
    m_in = mxgetm(prhs(1))
    n_in = mxgetn(prhs(1))
    pr_in = mxgetpr(prhs(1))
    plhs(1) = mxcreatefull(m_in, n_in, 0)
    pr_out = mxgetpr(plhs(1))
    call dbl_mat(%val(pr_out), %val(pr_in), m_in, n_in)
    return
end
```

```fortran
C
C This is the computational routine.
subroutine dbl_mat(out_mat, in_mat, m, n)
    integer m, n
    integer rr, cc
    real*8 out_mat(m, n), in_mat(m, n)
    do 20 rr=1, m
      do 10 cc=1, n
        out_mat(rr, cc) = 2*in_mat(rr, cc)
      10 continue
    20 continue
    return
end
```

For an input 2-by-3 matrix

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

typing

```
y = dbl_mat(x)
```

yields

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

Microsoft Fortran assumes a C language calling convention when %val is
invoked. Accordingly, if you want to use %val to call a Fortran subroutine, you
must modify the dbl_mat example from above to include extra gateway code
that converts the calling convention assumptions of %val back to Fortran con-
ventions. This code is the Microsoft Fortran version of dbl_mat:

C This subroutine is the main gateway to MATLAB.
C When a MEX function is executed MATLAB calls the USRFcn
C subroutine in the corresponding MEX file.
C DO NOT modify this subroutine declaration.
C
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C This is the library needed to set the floating point exception
C word and it needs to be here in the code.
use msflib
integer *4 nlhs, nrhs
integer plhs(*), prhs(*)
ingteger mxcreatefull, mxgetpr, mxalloc
integer mxGetM, mxGetN
real*8 mxGetScalar

C
C These are the interfaces needed to accommodate
C the use of %val when these functions are called.
C An Interface block needs to be added for any
C new subroutine that uses %val.
C
interface
subroutine dbl_mat_msf(mat_out, in_mat, m, n)
implicit double precision(A–H,O–Z)
ingteger m, n
end subroutine dbl_mat_msf
end interface

C
C These interfaces are not necessary for this program, but are
C included for convenient use in other user-built programs.
C
interface
subroutine mxCopyPtrToReal8(A, B, C)
real*8 B
integer A, C
end subroutine mxCopyPtrToReal8
end interface

C
interface
subroutine mxCopyReal8ToPtr(A, B, C)
real*8 B
integer A, C
end subroutine mxCopyReal8ToPtr
end interface

C User should modify the following code to fit his or her requirements.
integer m, n
integer pr_in, pr_out

C Set the floating-point control word to allow divide by zero.
Note: this is not used in dbl_mat_msf, but is necessary in situations where a divide by zero may occur. This is included here for convenience.
integer(2) control, newcontrol
call getcontrolfpqq(control)
newcontrol = control .or. FPCW$ZERODIVIDE
call setcontrolfpqq(newcontrol)

C Proceed with the Fortran gateway code.
m = mxGetM(prhs(1))
n = mxGetM(prhs(1))

C Check that input is numeric.
if(mxIsNumeric(prhs(1)) .ne. 1) then
call mexErrMsgTxt('Input is not numeric.')
endif
pr_in = mxgetpr(prhs(1))
plhs(1) = mxcreatefull(m, n, 0)
pr_out = mxgetpr(plhs(1))

C Call the computational subroutine.
call dbl_mat_msf(%val(pr_out), %val(pr_in), m, n)
return
end

C This is the computational routine. It is the same as the previous dbl_mat.
subroutine dbl_mat_msf(out_mat, in_mat, m, n)
  integer m, n
  integer rr, cc
  real*8 out_mat(m, n), in_mat(m, n)
  do 20 rr=1, m
    do 10 cc=1, n
      out_mat(rr, cc) = 2*in_mat(rr, cc)
    10     continue
  20   continue
return
end

You can use this example as a template for building your own MEX-files that invoke %val on the Microsoft Fortran compiler. Results for this function should agree with the results obtained in dbl_mat.f (described in the previous section).

Handling Sparse Matrices

The MATLAB 5 API provides a set of functions that allow you to create and manipulate sparse matrices from within your MEX-files. There are special parameters associated with sparse matrices, namely \texttt{ir}, \texttt{jc}, and \texttt{nzmax}. For information on how to use these parameters and how MATLAB stores sparse matrices in general, refer to “The MATLAB Array” section in Chapter 1 of this book.

This example reads in a sparse matrix from MATLAB and prints its contents to an ASCII file:
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer nlhs, nrhs, plhs(*), prhs(*)
integer nzmax, N, pr(10), ir(10), jc(10), xrows(10), xcols(10)
C Check to ensure input is sparse.
if(mxIsSparse(prhs(1) .eq. 0) then
    call mexErrMsgTxt('Input is not a sparse matrix.')
endif
C Get the data pointers.
    nzmax = mxGetNzmax(prhs(1))
    pr = mxGetPr(prhs(1))
    ir = mxGetIr(prhs(1))
    jc = mxGetJc(prhs(1))
C Copy the data and indices to Fortran matrices.
call mxCopyPtrToReal8(pr, x, nzmax)
call mxCopyPtrToInteger4(ir, xrows, nzmax)
call mxCopyPtrToInteger4(jc, xcols, nzmax)
C Call the computational subroutine.
call printsparse(x, xrows, xcols, nzmax)
return
end
C The computational subroutine.
subroutine printsparse(x, xrows, xcols, nzmax)
real*8 xcols(10)
integer nzmax, N, i, j, xrows(10), xcols(10)
C Open a file to write out the sparse matrix contents.
open(unit=8, file='sparse.dat', status='new')
do 10 i=1, nzmax
    write(8,111), xrows(i), xcols(i), x(i))
111 format('The contents of Row ',i3,' Column ',i3,' are
X ',F6.2)
do 10 continue
close(8)
return
end
Note that *mxCopyPtrToInteger4* is used to copy the data in vectors *ir* and *jc* to Fortran matrices. This is one of the only times when *mxCopyPtrToInteger4* can be used (the other use involves *matGetDir*; see Chapter 4, “Data Export and Import”). When passing an integer to a Fortran MEX-file, remember that MATLAB stores integers as double-precision, real numbers. This means that you must use *mxCopyPtrToReal8* to copy integers into a Fortran matrix.

For example, create a sparse 10-by-10 matrix in MATLAB:

```matlab
i = [1 2 4 8 9]; j = [2 3 5 7 9]; s = [1 2 3 4 5]; m = 10; n = 10;
S = sparse(i, j, s, m, n);
```

Typing
```
print sparse(S);
```
produces a file called *for008.dat* that contains the non-zero elements of *S*.

### Calling MATLAB Functions from Fortran MEX-Files

It's possible to call MATLAB functions, operators, M-files, and even other MEX-files from within your Fortran source code by using the API function *mexCallMATLAB*. This example forms a special matrix, calculates its eigenvalues and eigenvectors using the MATLAB *eig* function, and displays the inverse values.

```fortran
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
  integer nlhs, nrhs, plhs(*), prhs(*)
  call EIGS(nlhs, plhs)
  return
end
subroutine EIGS(nlhs, plhs)
  integer nlhs
  integer plhs(*)
  C-------------------------------------------------------------
  C This routine first forms and displays the matrix (in MATLAB notation):
  C
  C      hankel(1:4,4:-1:1) + sqrt(-1)*toeplitz(1:4,1:4)
  C
  C Next it finds the eigenvalues and eigenvectors (using the MATLAB function *EIG*), and displays the eigenvalue matrix. Then it calculates the inverse of the eigenvalues to demonstrate
```
Examples of Fortran MEX-Files

C manipulation of MATLAB results and how to deal with complex C arithmetic. Finally, the program displays the inverse values.

integer mxCreateFull, mxGetPr, mxGetPi
C---------------------------------------------------------------
C Declare all the variables we intend to use for the example.
    integer m, n, imagflag, mlhs, mrhs, mn
C---------------------------------------------------------------
    integer x, xr, xi, lhs(2), rhs
    integer evaluer, evaluei
C---------------------------------------------------------------
C Fill xreal and ximag arrays with the matrix (in MATLAB notation)
C hankel(1:4, 4:-1:1)+sqrt(-1)*toeplitz(1:4, 1:4)
    call fill(xreal, ximag)
    m = 4
    n = 4
    mn = m*n
    imagflag = 1
C Create (allocate) a complex matrix of size m-by-n
    x = mxCreateFull(m, n, imagflag)
C Get pointers to the real and imaginary parts of the matrix
    xr = mxGetPr(x)
    xi = mxGetPi(x)
C Copy our arrays xreal and ximag to their corresponding places in
C the matrix structure x.
    call mxCopyReal8ToPtr(xreal, xr, mn)
    call mxCopyReal8ToPtr(ximag, xi, mn)
C Find eigenvectors and eigenvalues of matrix x using the MATLAB
C function EIG. We give mexCallMATLAB one right-hand-argument
C and expect two left-hand-side arrays to be returned.
    mrhs = 1
    mlhs = 2
    call mexCallMATLAB(mlhs, lhs, mrhs, x, 'eig')
C Display original x matrix, therefore there is no left-hand
C side.
    mlhs = 0
    if (nlhs .eq. 0) then
        call mexCallMATLAB(mlhs, lhs, mrhs, x, 'disp')
    endif
C Display eigenvalue matrix.
rhs = lhs(2)
if (nlhs .eq. 0) then
  call mexCallMATLAB(mlhs, lhs, mrhs, rhs, 'disp')
endif
C Invert diagonal elements of the complex matrix by two methods.
C The first uses auxiliary arrays for performing the inversion,
C the second does the calculations in place.
evaluer = mxGetPr(lhs(2))
evaluei = mxGetPi(lhs(2))
C Copy these to new working arrays.
call mxCopyPtrToReal8(evaluer, xreal, mn)
call mxCopyPtrToReal8(evaluei, ximag, mn)
call invertd(xreal, ximag)
C Copy answers back to a MATLAB array x.
call mxCopyReal8ToPtr((xreal, xr, mn)
call mxCopyReal8ToPtr((ximag, xi, mn)
if (nlhs .eq. 0) then
  call mexCallMATLAB(mlhs, lhs, mrhs, x, 'disp')
endif
C Free all allocated matrices.
call mxFreeMatrix(x)
call mxFreeMatrix(lhs(2))
if (nlhs .ne. 0) goto 1000
call mxFreeMatrix(lhs(1))
goto 1010
1000 plhs(1) = lhs(1)
1010 return
end
C subroutine fill(xr, xi)
double precision xr(4,4), xi(4,4), tmp
integer i, j
C Form the matrix: hankel(1:4,4:-1:1)+sqrt(-1)*toeplitz(1:4,1:4)
do 1000 j=1,4
  do 1000 i=1,j
    xr(i,j) = 4+i-j
    xr(j,i) = xr(i,j)
    xi(i,j) = j-i+1
  enddo
1000 xi(i,i) = xi(i,i)
C Reorder columns of xr.
do 1050 j=1,2
do 1050 i=1,4
    tmp = xr(i,j)
    jj = 5-j
    xr(i,jj) = xr(i,jj)
1050   xr(i,jj) = tmp
return
end

C subroutine invertd(xr, xi)
double precision xr(*), xi(*), tmp
integer i
C Just invert diagonal elements of a complex matrix of order 4.
C
do 1000 i=1,16,5
    tmp = xr(i)**2+xi(i)**2
    xr(i) = xr(i)/tmp
1000   xi(i) = –xi(i)/tmp
return
end

C subroutine copyrl(x, y, n)
C Copy one length n real array to another.
C
double precision x(*), y(*)
integer n, i

do 1000 i=1,n
1000   y(i) = x(i)
return
end

It is possible to use mexCallMATLAB (or any other API routine) from within your computational Fortran subroutine. Note that you can only call most MATLAB functions with double-precision data. M-functions, like eig, will not work with data that is of different precision than double.
Advanced Topics

This section covers advanced features of MEX-files that you can use when your applications require sophisticated MEX-files.

Help Files

Because the MATLAB interpreter chooses the MEX-file when both an M-file and a MEX-file having the same name are encountered in the same directory, it is possible to use M-files for documenting the behavior of your MEX-files. The MATLAB help command will automatically find and display the appropriate M-file when help is requested and the interpreter will find and execute the corresponding MEX-file when the function is actually invoked.

Linking Multiple Files

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

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

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

You may find it useful to use a software development tool like MAKE to manage MEX-file projects involving multiple source files. Simply create a MAKEFILE that contains a rule for producing object files from each of your source files and then invokes mex to combine your object files into a MEX-file. This way you can ensure that your source files are recompiled only when necessary.

Variable Scope

Being inside a MEX-file is much like being inside an M-file function. However, there are two important differences regarding the scope of variables between the behavior of M-files and MEX-files:

- MEX-files can have static (persistent) variables; M-files cannot.
- MEX-files can modify variables in the caller's workspace; M-files cannot.

MEX-files can use the routines mexPutArray and mexEvalString if they need to modify the caller's workspace.
How to Debug Fortran Language MEX-Files

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

**Note:** The sections, “Troubleshooting MEX-File Creation” and “Understanding MEX-File Problems,” in Chapter 6 provide additional information on isolating problems with MEX-files.

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

```
mex -g filename.f
```

**Debugging on UNIX**

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

```
matlab -D dbx
```

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

```
dbmex on
```

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

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

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

```
  nm <MEX-file> | grep function
```

To proceed from the breakpoint, issue a “continue” command to the debugger. Once you hit one of your breakpoints, you can make full use of any facilities your debugger provides to examine variables, display memory, or inspect registers. Refer to the documentation provided with your debugger for information on its use.

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

```
  dbmex stop
```

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

### Debugging on Windows

**Microsoft Compiler.** If you are using the Microsoft compiler:

1. Start MATLAB in the debugger by typing on the command line

   `MSDEV -debug MATLAB.EXE`

2. Open your MEX-file source code.

3. Set a breakpoint on the desired line of code by right-clicking your mouse on the line of code.

4. From the **Build** menu, select **Debug** option, and click **GO**.
Debugging on Macintosh

CodeWarrior Compiler. If you are using the CodeWarrior compiler on Power Macintosh or 68K Macintosh systems:

1. Build the MEX-file from MATLAB using `mex.m`. In addition to the normal switches you use to build the MEX-file, specify the `-g` switch.

2. Start the Metrowerks Debugger application. When it asks you to select a `SYM` file, select the `.xSYM` file created by CodeWarrior when you built the MEX-file (e.g., `timestwo.mex` would have a `SYM` file called `timestwo.mex.xSYM`).

3. Set breakpoint(s) within your MEX-file's source code.

4. Run the MEX-function from within MATLAB.

The Metrowerks Debugger will automatically pop to the foreground when a breakpoint is reached.

Jasik Debugger on 68K. If you are using the Jasik Debugger on MEX-files created with MPW SC, LS Fortran, or CodeWarrior C on 68K Macintosh systems:

1. Create a `.dsi` file called `MATLAB.dsi` with any text editor (such as MPW or SimpleText). It should look similar to the following example, and contain at least the following information:

```
=GMEX0; Debug code resources of type 'MEX0' (M-E-X-zero)
=FDbg_Rsrcs = 1; Enable resource debugging
=E; End .dsi file
```

2. Place the `MATLAB.dsi` file in the same folder as the MATLAB binary.
3 You will need a dummy .MAP file so that the debugger will open and execute the MATLAB.dsi file. Place the following in a text file called MATLAB.MAP and put this file into the same folder as the MATLAB binary:

MATLAB.MAP

```
Segment 'Main' size=$000000 rsrcid=0 Jtindex=$0000 #TEnts=$0000
__Startup__                        $000000  size=$000000  extern
```

4 Build the MEX-file from MATLAB (using mex.m) or MPW (using mex). In addition to the normal switches you use to build the MEX-file, specify the -g switch.

5 Rename the resulting .SYM file from <filename>.SYM to <filename>/MEX0_0.SYM. For example, conv.mex.SYM becomes conv.mex/MEX0_0.SYM (MEX0_0 is M-E-X-zero-underscore-zero). Make sure this file is in the same folder as the MEX-file.

6 Create a .dsi file for the MEX-file itself. It should look similar to the following example, and contain at least the following information:

```
<M filename>/MEX0_0.dsi

=bkpt
mexFunction ; set breakpoint at function mexFunction
=5
MyDisk:myFirstSourceFolder; ; set list of folders where source
     ; files reside
MyDisk:mySecondSourceFolder:
=E
; End .dsi file
```
7 Name this file `<filename>/MEX0_0.dsi`. Using the `conv` example above, this becomes `conv.mex/MEX0_0.dsi`. Make sure this file is in the same folder as the MEX-file.

8 Run the MEX-function from within MATLAB. The debugger will automatically take control when the MEX-function is called, executing its `.dsi` file and stopping at the breakpoint for `mexFunction`.

9 If the debugger asks you if you want to set Source Paths, choose No. Step through a few lines of assembly code until you reach the instruction

```
BSR mexFunction
```

10 Step into this instruction to go to the Fortran function `mexFunction`.

**Jasik Debugger on Power Macintosh.** If you are using the Jasik Debugger on MEX-files created with MPW MrC, LS Fortran, or CodeWarrior C on Power Macintosh systems:

1 From within the debugger, make sure the `Dbg_SLM` option in the `–Dbgr Status` window is set to ON.

2 Build the MEX-file from MATLAB (using `mex.m`) or MPW (using `mex`). In addition to the normal switches you use to build the MEX-file, specify the `-g` switch.

3 Rename the resulting `.xSYM` file to `mexFunction.xSYM`. Make sure this file is in the same folder as the MEX-file.
4 Create a .dsi file for the MEX-file itself. It should look similar to the following example, and contain at least the following information:

```plaintext
mexFunction.dsi

=bkpt
mexFunction; set breakpoint at function mexFunction

=5
MyDisk:myFirstSourceFolder: ; set list of folders where source files reside
MyDisk:mySecondSourceFolder:

=E
; End .dsi file
```

5 Name this file `mexFunction.dsi`, and make sure it is in the same folder as the MEX-file.

6 Run the MEX-function from within MATLAB. The debugger will automatically take control when the MEX-function is called, executing its .dsi file and stopping at the breakpoint for `mexFunction`. 
Data Export and Import

4-2 Using MAT-Files
4-2 Importing Data to MATLAB
4-3 Exporting Data from MATLAB
4-4 Exchanging Data Files Between Platforms
4-5 Reading and Writing MAT-Files
4-7 Directory Organization

4-11 Examples of MAT-Files
4-11 Writing a MAT-File
4-13 Reading a MAT-File

4-15 Compiling and Linking MAT-File Programs
4-15 UNIX
4-15 Windows
4-16 Macintosh
Using MAT-Files

This section describes the various techniques for importing data to and exporting data from the MATLAB environment. The most important approach involves the use of MAT-files—the data file format that MATLAB uses for saving data to your disk. MAT-files provide a convenient mechanism for moving your MATLAB data between different platforms in a highly portable manner as well as for importing and exporting your data to other stand-alone MATLAB applications. To simplify your use of MAT-files in applications outside of MATLAB, we have developed a library of access routines with a `mat` prefix that you can use in your own C or Fortran programs to read and write MAT-files. Programs that access MAT-files also use the `mx` prefixed API routines discussed in the “Creating C Language MEX-Files” and “Creating Fortran MEX-Files” chapters of this book.

This chapter includes these topics:

- Importing data to MATLAB
- Exporting data from MATLAB
- How to read and write to MAT-files
- Examples of writing to and reading from MAT-files

Finally, because linking and compiling programs that read or write to MAT-files is somewhat different on the Macintosh than on Windows or UNIX, this chapter ends with a discussion of how to compile and link on both the Power Mac and 68K Macintosh systems.

Importing Data to MATLAB

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

- Enter the data as an explicit list of elements. If you have a small amount of data, less than 10-15 elements, it is easy to type the data explicitly using brackets `[ ]`. This method is awkward for larger
amounts of data because you can’t edit your input if you make a mistake.

• Create data in an M-file. Use your text editor to create an M-file that enters your data as an explicit list of elements. This method is useful when the data isn’t already in computer-readable form and you have to type it in. Essentially the same as the first method, this method has the advantage of allowing you to use your editor to change the data and correct mistakes. You can then just rerun your M-file to re-enter the data.

• Load data from an ASCII flat file. A flat file stores the data in ASCII form, with fixed-length rows terminated with new lines (carriage returns) and with spaces separating the numbers. You can edit ASCII flat files using a normal text editor. Flat files can be read directly into MATLAB using the `load` command. The result is to create a variable with the same name as the filename.

• Read data using `fopen`, `fread`, and MATLAB’s other low-level I/O functions. This method is useful for loading data files from other applications that have their own established file formats.

• Write a MEX-file to read the data. This is the method of choice if subroutines are already available for reading data files from other applications. See the section, “Dynamically Linked Subroutines: MEX-Files,” for more information.

• Write a program in C or Fortran to translate your data into MAT-file format and then read the MAT-file into MATLAB with the `load` command. Refer to the section, “Reading and Writing MAT-Files,” for more information.

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

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

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

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

creates an ASCII file called `temp.dat` containing:

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

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

- Write the data in a special format using `fopen`, `fwrite`, and the other low-level I/O functions. This method is useful for writing data files in the file formats required by other applications.

- Develop a MEX-file to write the data. This is the method of choice if subroutines are already available for writing data files in the form needed by other applications. See the section, “Dynamically Linked Subroutines: MEX-Files,” for more information.

- Write out the data as a MAT-file using the `save` command, and then write a program in C or Fortran to translate the MAT-file into your own special format. See the section, “Reading and Writing MAT-Files,” for more information.

### Exchanging Data Files Between Platforms

You may want to work with MATLAB implementations on several different computer systems, or need to transmit MATLAB applications to users on other systems. MATLAB applications consist of M-files containing functions and scripts, and MAT-files containing binary data. Both types of files can be transported directly between machines because they contain a machine signature in the file header. MATLAB checks the signature when it loads a file and, if a signature indicates that a file is foreign, performs the necessary conversion.
Using MATLAB across several different machine architectures requires a facility for exchanging both binary and ASCII data between the various machines. Examples of this type of facility include FTP, NFS, Kermit, and other communication programs. When using these programs, be careful to transmit binary MAT-files in binary file mode and ASCII M-files in ASCII file mode. Failure to set these modes correctly with these programs corrupts the data.

**Reading and Writing MAT-Files**

The `save` command in MATLAB saves the MATLAB arrays currently in memory to a binary disk file called a MAT-file. The term MAT-file is used because these files have the extension `.mat`. The `load` command performs the reverse operation: it reads the MATLAB arrays from a MAT-file on disk back into MATLAB’s workspace.

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

**MAT-File Subroutine Library**

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

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

<table>
<thead>
<tr>
<th>MAT-Function (C and Fortran)</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matOpen</code></td>
<td>Open a MAT-file</td>
</tr>
<tr>
<td><code>matClose</code></td>
<td>Close a MAT-file</td>
</tr>
<tr>
<td><code>matGetDir</code></td>
<td>Get a list of MATLAB arrays from a MAT-file</td>
</tr>
</tbody>
</table>
### MAT-Function (Fortran only)
<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matGetFull</code></td>
<td>Read a full <code>mxArray</code> from a MAT-file</td>
</tr>
<tr>
<td><code>matPutFull</code></td>
<td>Write a full <code>mxArray</code> to a MAT-file</td>
</tr>
<tr>
<td><code>matGetMatrix</code></td>
<td>Get a named MATLAB array from a MAT-file</td>
</tr>
<tr>
<td><code>matPutMatrix</code></td>
<td>Put a MATLAB array into a MAT-file</td>
</tr>
<tr>
<td><code>matGetNextMatrix</code></td>
<td>Get the next sequential MATLAB array from a MAT-file</td>
</tr>
<tr>
<td><code>matDeleteMatrix</code></td>
<td>Remove a MATLAB array from a MAT-file</td>
</tr>
<tr>
<td><code>matGetString</code></td>
<td>Read a string <code>mxArray</code> from a MAT-file</td>
</tr>
<tr>
<td><code>matPutString</code></td>
<td>Write a string <code>mxArray</code> to a MAT-file</td>
</tr>
</tbody>
</table>

### MAT-Function (C only)
<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matGetFp</code></td>
<td>Get an ANSI C file pointer to a MAT-file</td>
</tr>
<tr>
<td><code>matGetArray</code></td>
<td>Read an <code>mxArray</code> from a MAT-file</td>
</tr>
<tr>
<td><code>matPutArray</code></td>
<td>Write an <code>mxArray</code> to a MAT-file</td>
</tr>
<tr>
<td><code>matGetNextArray</code></td>
<td>Read the next <code>mxArray</code> from a MAT-file</td>
</tr>
<tr>
<td><code>matDeleteArray</code></td>
<td>Remove an <code>mxArray</code> from a MAT-file</td>
</tr>
</tbody>
</table>
Using MAT-Files

MAT-Function (C only) | Purpose
--- | ---
matPutArrayAsGlobal | Put a MATLAB array into a MAT-file such that the load command will place it into the global workspace
matGetArrayHeader | Load a MATLAB array header from a MAT-file
matGetNextArrayHeader | Load only the next MATLAB array header from a MAT-file

Directory Organization
A collection of files associated with reading and writing MAT-files is located on your disk. Below is a table listing the path to the subdirectories that you need to import and export data using MAT-functions:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Directories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td><code>&lt;matlab&gt;\extern\include</code> \bin \examples</td>
</tr>
<tr>
<td>UNIX Workstation</td>
<td><code>$MATLAB/extern/include</code> /lib /examples</td>
</tr>
<tr>
<td>Macintosh</td>
<td><code>&lt;MATLAB&gt;:extern:include</code> :lib :examples</td>
</tr>
</tbody>
</table>
The `include` directory holds header files containing function declarations with prototypes for the routines that you can access in the API Library. Included in the subdirectory are:

- `matrix.h`, the header file that defines MATLAB array access method prototypes
- `mat.h`, the header file that defines MAT-file access method prototypes
The bin subdirectory contains shared (dynamically linkable) libraries for linking your programs.

<table>
<thead>
<tr>
<th>Windows</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libmat.dll</td>
<td>The library of MAT-file routines (C only)</td>
</tr>
<tr>
<td>libmx.dll</td>
<td>The library of array access routines</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UNIX</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libmat.so</td>
<td>The library of MAT-file routines (C and Fortran)</td>
</tr>
<tr>
<td>libmx.so</td>
<td>The library of array access routines</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Macintosh</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>68K: MPW:libmat.o</td>
<td>The library of MAT-file routines for use with MPW SC on 68K platforms</td>
</tr>
<tr>
<td>68K: MPW:libmx.o</td>
<td>The library of array access routines for use with MPW SC on 68K platforms</td>
</tr>
<tr>
<td>68K:Metrowerks:libmat.lib</td>
<td>The library of MAT-file routines for use with Metrowerks C on 68K platforms</td>
</tr>
<tr>
<td>68K:Metrowerks:libmx.lib</td>
<td>The library of array access routines for use with Metrowerks C on 68K platforms</td>
</tr>
<tr>
<td>libmat</td>
<td>The library of MAT-file routines for use with a PowerPC compiler</td>
</tr>
<tr>
<td>libmx</td>
<td>The library of array access routines for use with a PowerPC compiler</td>
</tr>
</tbody>
</table>
The examples subdirectory contains C and Fortran source code for a number of example files that demonstrate how to use the MAT-file routines:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matdemo.c</td>
<td>Example C program that demonstrates the use of MAT-file routines</td>
</tr>
<tr>
<td>matdemo1.f</td>
<td>matdemo2.f</td>
</tr>
</tbody>
</table>

This table lists the path to the example files on several platforms:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td><code>&lt;matlab&gt;\extern\examples\eng_mat</code></td>
</tr>
<tr>
<td>UNIX</td>
<td><code>$MATLAB/extern/examples/eng_mat</code></td>
</tr>
<tr>
<td>Macintosh</td>
<td><code>&lt;matlab&gt;:extern:examples:eng_mat</code></td>
</tr>
</tbody>
</table>
Examples of MAT-Files

The next sections of this chapter include examples of writing and reading MAT-files.

Writing a MAT-File

Here is a simple C program illustrating how to use the library routines to create a MAT-file. This example writes a 3-by-2 real array into a MAT-file, naming the array A and the MAT-file dataout.mat:

```c
#include <string.h>
#include "mat.h"
static double Areal[6]={1, 2, 3, 4, 5, 6};
main()
{
    MATFile *fp;
    mxArray *a;
    fp = matOpen("dataout.mat", "w");
    a = mxCreateDoubleMatrix(3, 2, mxREAL);
    memcpy(mxGetPr(a), Areal, 6*sizeof(double));
    mxSetName(a, "A");
    matPutArray(fp, a);
    matClose(fp);
    mxDestroyArray(a);
}
```

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

Once you have compiled and linked your MAT-file program, you can run the executable you have just produced. This example creates the file dataout.mat, which you can examine by using MATLAB. Start up MATLAB and enter:

```
load dataout
A
```
You should see:

\[
A = 
\begin{pmatrix}
1 & 4 \\
2 & 5 \\
3 & 6
\end{pmatrix}
\]

For an example of a Windows stand-alone program, see `engwindemo.c`.

Alternatively, this example could have been coded in Fortran, in which case it would look like this:

```fortran
program main
integer matOpen, mxCreateFull, matClose
integer mxGetPr, matPutMatrix
integer a, fp, stat
double precision Areal(6)
data Areal / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 /

fp = matopen("dataout.mat", "w")
a = mxCreateFull(3, 2, 0)
call mxCopyReal8toPtr(Areal, mxGetPr(a), 6)
call mxSetName(a, "A")
stat = matPutMatrix(fp, a)
stat = matClose(fp)
call mxFreeMatrix(a)
stop
end
```

**Note:** Fortran MAT-files are not available on Windows.
Reading a MAT-File

The next example illustrates how to read the array A from the MAT-file created with the previous example into an array of doubles.

```c
#include <string.h>
#include <stdio.h>
#include "mat.h"
static double Areal[6];
main()
{
    MATFile *fp;
    mxArray *a;
    int mn;
    if (!(fp = matOpen("dataout.mat", "r"))
        fprintf(stderr, "Can't open file.");
    a = matGetArray(fp, "A");
    mn = mxGetM(a) * mxGetN(a);
    memcpy(Areal, mxGetPr(a), mn*sizeof(double));
    matClose(fp);
    mxDestroyArray(a);
    mexPrintf("Row 1: %g %g", Areal[0], Areal[3]);
}
```

After you compile, link, and run this file, it should print the first row of array A.
A Fortran version of this routine is:

```fortran
program main
  integer matOpen, matGetMatrix, matOpen, matClose
  integer mxGetM, mxGetN
  integer a, fp, mn, stat

  fp = matopen("dataout.mat", "r")
  if(fp .eq. 0) then
    write(6,*), "Can't open file."
    stop
  endif
  a = matGetMatrix(fp, "A")
  mn = mxGetM(a) * mxGetN(a)
  call mxCopyPtrtoReal8(mxGetPr(a), Areal, 6)
  stat = matClose(fp)
  call mxFreeMatrix(a)
  write(6,*), "Row 1: ", Areal(1), ", Areal(4)
  stop
end
```

After you compile, link, and run this file, it should also print the first row of array A.
Compiling and Linking MAT-File Programs

UNIX

This table lists the steps required to compile and link MAT-file C and Fortran programs on UNIX platforms:

<table>
<thead>
<tr>
<th>UNIX - C</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP700</td>
<td><code>cc -Aa &lt;include dir&gt; -o &lt;result&gt; &lt;source&gt; &lt;libdir&gt; &lt;libraries&gt;</code></td>
</tr>
<tr>
<td>Linux</td>
<td><code>gcc -ansi &lt;include dir&gt; -o &lt;result&gt; &lt;source&gt; &lt;libdir&gt; &lt;libraries&gt;</code></td>
</tr>
<tr>
<td>SunOS 4.x</td>
<td><code>acc &lt;include dir&gt; -o &lt;result&gt; &lt;source&gt; &lt;libdir&gt; &lt;libraries&gt;</code></td>
</tr>
<tr>
<td>All others</td>
<td><code>cc &lt;include dir&gt; -o &lt;result&gt; &lt;source&gt; &lt;libdir&gt; &lt;libraries&gt;</code></td>
</tr>
</tbody>
</table>

UNIX - Fortran

<table>
<thead>
<tr>
<th>UNIX - Fortran</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td><code>f77 &lt;include dir&gt; -o &lt;result&gt; &lt;source&gt; &lt;libdir&gt; &lt;libraries&gt;</code></td>
</tr>
</tbody>
</table>

where:

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

Windows

Fortran MAT-file support is currently unavailable on Windows. To compile and link C MAT-file programs, use the `mex` script with a MAT options file. The files, `watengmatopts.bat`, `bccengmatopts.bat`, and `msvcengmatopts.bat` are stand-alone MAT and engine options files (located in `MATLAB/bin`). Refer to them for details on how to customize a MAT options file for your particular compiler.
Macintosh
Compiling and linking MAT-file C and Fortran programs on Macintosh is somewhat different than on other platforms. This section discusses some general information you need to know and then discusses how to build applications that use the MAT API in these environments:

- Metrowerks C on the Power Macintosh
- MPW C on the Power Macintosh
- Metrowerks C on 68K Macintosh
- MPW C on 68K Macintosh
- Language Systems Fortran on the Power Macintosh
- Language Systems Fortran on 68K Macintosh

General Macintosh Information
To use the MAT API on a Power Macintosh, the four shared libraries libut, libmx, libmi, and libmat must be in either the same folder as the application you create or in the Extensions folder in the System Folder. A simple way to accomplish this is to place an alias of the <MATLAB>:extern:lib:PowerMac: folder (where the shared libraries reside) in the Extensions folder. You may want to rename the alias to something like "MATLAB shared libraries."

Metrowerks C on the Power Macintosh
To build applications that use the MAT API, you must make these changes to a project initialized from the ~ANSI PPC C/C++ project stationery provided in the <MATLAB>:extern:src: folder:

- In the Access Paths preference panel, add the folder <MATLAB>:extern:include: to the User field.
- In the C/C++ Language preference panel, check the Enums Always Int box.
- Add the source file(s) that call MAT API functions to the project.
- Add the shared libraries libmx and libmat from the folder <MATLAB>:extern:lib:PowerMac: to the project.

You can now build your application following the same steps you would use to build any application in CodeWarrior.
MPW C on the Power Macintosh
To build applications that use the MAT API, you must follow these steps in addition to the normal steps required to build applications under MPW:

- Add the folder `<MATLAB>:extern:include:` to the include file search path when compiling the source files that call MAT API functions.
- Specify the `-enum int` switch when compiling the source files that call MAT API functions.
- Link the application against the shared libraries `libmx` and `libmat` from the `<MATLAB>:extern:lib:PowerMac:` folder.

For example:

```bash
set MATLABLibraries '{MATLABRoot}extern:lib:PowerMac:'
set MATLABIncDir '{MATLABRoot}extern:include:'
MrC mattest1.c -o mattest1.c.x -i '{MATLABIncDir}' -enum int
PPCLink mattest1.c.x -o mattest1 -t APPL -c '????'
    '{MATLABLibraries}libmx' ∂
    '{MATLABLibraries}libmat' ∂
    '{PPCLibraries}PPCRuntime.o' ∂
    '{PPCLibraries}StdCRunTime.o' ∂
    '{SharedLibraries}StdCLib' ∂
```

Metrowerks C on 68K Macintosh
To build applications that use the MAT API, make the following changes to a project initialized from the `~ANSI Mac OS 4i8d/F 68k C/C++` project stationery provided in the `<MATLAB>:extern:src:` folder:

- In the Access Paths preference panel, add the folder `<MATLAB>:extern:include:` to the User field.
- In the 68K Processor preference dialog, check the box marked 68020 CodeGen.
- In the C/C++ Language preference panel, check the Enums Always Int box.
- Add the source file(s) that call MAT API functions to the project.
• Add the static libraries `libmx.lib`, `libmi.lib`, `libut.lib`, and `libmat.lib` from the `<MATLAB>:extern:lib:68k:Metrowerks:folder` to the project.

You can now build your application following the same steps you would use to build any application in CodeWarrior.

**MPW C on 68K Macintosh**
To build applications that use the MAT API, do the following steps in addition to the normal steps required to build applications under MPW:

• Add the folder `<MATLAB>:extern:include:` to the `include` file search path when compiling the source files that call MAT API functions.

• Specify the `-mc68020` and `-elems881` switches to use 68020 code generation and 68881 coprocessor code generation, when compiling the source files that call MAT API functions.

• Specify the `-enum int` switch when compiling the source files that call MAT API functions.

• Specify the `-model far` switch when linking the application.

• Link the application against the static libraries `libmx.o`, `libmi.o`, `libut.o`, and `libmat.o` from the `<MATLAB>:extern:lib:68k:MPW:folder`. 
Compiling and Linking MAT-File Programs

For example:

```bash
set MATLABLibraries "{MATLABRoot}extern:lib:68k:MPW:
set MATLABIncDir "{MATLABRoot}extern:include:
SC mattest1.c -o mattest1.c.o -mc68020 -elems881 -i
  "{MATLABIncDir}" -enum int
Link mattest1.c.o -o mattest1 -t APPL -c '????' -
  -model far -w -
  {MATLABLibraries}libmx.o -
  {MATLABLibraries}libut.o -
  {MATLABLibraries}libmi.o -
  {CLibraries}CLib881.o -
  {CLibraries}MathLib881.o -
  {CLibraries}StdCLib.o -
  {Libraries}MacRuntime.o -
  {Libraries}IntEnv.o -
  {Libraries}ToolLibs.o -
  {Libraries}Interface.o -
```

Language Systems Fortran on the Power Macintosh

To build applications that use the MAT API, you must follow these steps in addition to the normal steps required to build applications under Language Systems Fortran:

- Link the application against the shared libraries `libmx` and `libmat` from the `<MATLAB>: extern:lib:PowerMac:` folder.

For example:

```bash
set MATLABLibraries "{MATLABRoot}extern:lib:PowerMac:
set MATLABIncDir "{MATLABRoot}extern:include:
FORTRAN.PPC 'mattest1.f' -o mattest1.f.x -opt=0
PPCLink -w -
  -t APPL -c '????'
  'mattest1.f.x' -
  "{PPCLibraries}"PPCCRuntime.o -
  "{PPCLibraries}"StdCRuntime.o -
  "{SharedLibraries}"StdLib -
  "{SharedLibraries}"MathLib -
  "{SharedLibraries}"InterfaceLib -
```
Language Systems Fortran on 68K Macintosh

To build applications that use the MAT API, follow these steps in addition to the normal steps required to build applications under Language Systems Fortran:

- Specify the `-mc68020` and `-FPU` switches to use 68020 code generation and 68881 coprocessor code generation, when compiling the source files that call MAT API functions.
- Specify the `-model far` switch when linking the application.
- Link the application against the static libraries `libmx.o`, `libmi.o`, `libut.o`, and `libmat.o` from the `<MATLAB>:extern:lib:68k:MPW:` folder.
- Link the application against the static libraries `CLib881.far.o`, `MathLib881.far.o`, `StdClib.far.o`, and `IntEnv.far.o`, also from the `<MATLAB>:extern:lib:68k:MPW:` folder.

For example:

```bash
set MATLABLibraries "\{MATLABRoot\}extern:lib:68k:MPW:"
set MATLABIncDir "\{MATLABRoot\}extern:include:"
FORTRAN 'mattest1.f' @
    -o mattest1.f.o -opt=1 -mc68020 -FPU
Link -t APPL -c '????' -f -srt -ad 4 -w -model far @
    'mattest1.f.o' @
    '{Libraries}'RunTime.o @
    '{Libraries}'Interface.o @
    '{FLibraries}'FORTRANlib.o @
    '{FLibraries}'IntrinsicLibFPU.o @
    '{FLibraries}'FSANELibFPU.o @
    '{MATLABLibraries}'libmx.o @
```
"{MATLABLibraries}"libut.o ∂
"{MATLABLibraries}"libmi.o ∂
"{MATLABLibraries}"libmat.o ∂
"{MATLABLibraries}"CLib881.far.o ∂
"{MATLABLibraries}"MathLib881.far.o ∂
"{MATLABLibraries}"StdLib.far.o ∂
"{MATLABLibraries}"IntEnv.far.o ∂
-o mattest1 ∂

Echo "Include ∂"{FLibraries}Fresources.r∂" " > ∂
"{FLibraries}Resource.inc" ∂

Rez "{FLibraries}Resource.inc" -a -m -o "mattest1" ∂
FSIZE "mattest1"
Using the MATLAB Engine

5-2 Interprocess Communication: The MATLAB Engine
5-3 Example: Calling the MATLAB Engine from a C Program

5-5 Compiling and Linking Engine Programs
5-5 Windows
5-5 UNIX
5-7 Example: Calling the MATLAB Engine from a Fortran Program

5-8 ActiveX Automation for Windows
5-9 MATLAB ActiveX Automation Methods
5-12 Additional ActiveX Information

5-14 Dynamic Data Exchange (DDE)
5-14 DDE Concepts and Terminology
5-16 Accessing MATLAB as a Server
5-21 Using MATLAB as a Client
5-23 DDE Advisory Links
**Interprocess Communication: The MATLAB Engine**

The MATLAB Engine Library is a set of routines that allows you to call MATLAB from your own programs, thereby employing MATLAB as a computation engine. Some of the things you can do with the MATLAB engine are

- Call a math routine to invert an array or to compute an FFT from your own program. When employed in this manner, MATLAB is a powerful and programmable mathematical subroutine library.
- Build an entire system for a specific task, for example, radar signature analysis or gas chromatography, where the front end (GUI) is programmed in C and the back end (analysis) is programmed in MATLAB, thereby shortening development time.

The MATLAB engine operates by running in the background as a separate process from your own program. This offers several advantages:

- On UNIX, the MATLAB engine can run on your machine, or on any other UNIX machine on your network, including machines of a different architecture. Thus you could implement a user interface on your workstation and perform the computations on a faster machine located elsewhere on your network. See the `engOpen` reference page for further information.
- Instead of requiring that all of MATLAB be linked to your program (a substantial amount of code), only a small engine communication library is needed.
The engine library contains the following routines for controlling the MATLAB computation engine. Their names all begin with the three-letter prefix `eng`:

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>engOpen</code></td>
<td>Start up MATLAB engine</td>
</tr>
<tr>
<td><code>engClose</code></td>
<td>Shut down MATLAB engine</td>
</tr>
<tr>
<td><code>engGetArray</code></td>
<td>Get a MATLAB array from the MATLAB engine</td>
</tr>
<tr>
<td><code>engPutArray</code></td>
<td>Send a MATLAB array to the MATLAB engine</td>
</tr>
<tr>
<td><code>engEvalString</code></td>
<td>Execute a MATLAB command</td>
</tr>
<tr>
<td><code>engOutputBuffer</code></td>
<td>Create a buffer to store MATLAB text output</td>
</tr>
</tbody>
</table>

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

On UNIX, the Engine Library communicates with the MATLAB engine using pipes, and, if needed, `rsh` for remote execution. On Windows, the Engine Library communicates with MATLAB using ActiveX. The Engine Library is not currently available on the Macintosh.

**Example: Calling the MATLAB Engine from a C Program**

Here is a simple program that illustrates how to call the Engine Library from a C program. The example starts a MATLAB engine process, sends a 3-by-2 real array to it, computes the eigenvalues of the array multiplied...
by its transpose, gets the array back to the C program, and prints out the section eigenvalue:

```c
#include <stdio.h>
#include "engine.h"
static double Areal[6]={1, 2, 3, 4, 5, 6}

void main()
{
    Engine *ep;
    mxArray *a, *d;
    double *Dreal, *Dimag;

    a = mxCreateDoubleMatrix(3, 2, mxREAL);
    memcpy(mxGetPr(a), Areal, 6*sizeof(double));
    mxSetName(a, "A");
    if (!(ep = engOpen("\0"))
        printf("Can't start up MATLAB engine.");
    engPutArray(ep, a);
    engEvalString(ep, "d=eig(A*A')");
    d = engGetArray(ep, "d");
    engClose(ep);

    Dreal = mxGetPr(d);
    Dimag = mxGetPi(d);

    if(Dimag)
        printf("Eigval 2: \%g+%gi\n", Dreal[1], Dimag[1]);
    else
        printf("Eigval 2: \%g\n", Dreal[1]);

    mxDestroyArray(a);
    mxDestroyArray(d);
}
```

For the Windows version of this program, see `engwindemo.c`. Engine examples, like the MAT examples, are located in the `eng_mat` directory.
Compiling and Linking Engine Programs

To produce an executable version of an engine program, you must compile it and link it with the appropriate library.

**Windows**

To compile and link engine programs, use the `mex` script with an engine options file. `watengmatopts.bat`, `bccengmatopts.bat`, and `msvcengmatopts.bat` are stand-alone engine and MAT options files (located in `MATLAB\bin`).

**UNIX**

Under UNIX, you must tell the system where the API shared libraries reside. This table provides the necessary UNIX command:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP700</td>
<td><code>setenv SHLIB_PATH $MATLAB/extern/lib/hp700</code></td>
</tr>
<tr>
<td>IBM RS/6000</td>
<td><code>setenv LIBPATH $MATLAB/extern/lib/ibm_rs</code></td>
</tr>
<tr>
<td>All others</td>
<td><code>setenv LD_LIBRARY_PATH $MATLAB/extern/lib/$Arch</code></td>
</tr>
</tbody>
</table>

where:

- `$MATLAB` is the MATLAB root directory
- `$Arch` is your architecture

It is often convenient to place this command in a startup script such as `.cshrc.mine`. Then, the system will be able to locate these shared libraries automatically, and you will not have to re-issue the command at the start of each login session.
This table shows the commands required to compile and link engine programs on UNIX platforms:

<table>
<thead>
<tr>
<th>UNIX - C</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP700</td>
<td><code>cc -Aa &lt;include dir&gt; -o &lt;result&gt; &lt;source&gt; &lt;libdir&gt; &lt;libraries&gt;</code></td>
</tr>
<tr>
<td>Linux</td>
<td><code>gcc -ansi &lt;include dir&gt; -o &lt;result&gt; &lt;source&gt; &lt;libdir&gt; &lt;libraries&gt;</code></td>
</tr>
<tr>
<td>SunOS 4.x</td>
<td><code>acc &lt;include dir&gt; -o &lt;result&gt; &lt;source&gt; &lt;libdir&gt; &lt;libraries&gt;</code></td>
</tr>
<tr>
<td>All others</td>
<td><code>cc &lt;include dir&gt; -o &lt;result&gt; &lt;source&gt; &lt;libdir&gt; &lt;libraries&gt;</code></td>
</tr>
</tbody>
</table>

UNIX - Fortran

| All             | `f77 <include dir> -o <result> <source> <libdir> <libraries>`           |

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

You can now run the executable you have just produced.
Example: Calling the MATLAB Engine from a Fortran Program

A Fortran version of the previous example is listed below. (Fortran engine is not available on Windows.)

```fortran
program main
  integer ep, a, d, stat
  integer engOpen, engClose, engPutArray, engGetArray
  integer engEvalString
  integer mxGetPr
  double precision Areal(6), darray(3)
  data Areal / 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 /

  a = mxCreateArray(3, 2, 0);
  call mxCopyReal8ToPtr(Areal, mxGetPr(a), 6)
  call mxSetName(a, "A")
  memcpy(mxGetPr(a), Areal, 6*sizeof(double));
  mxSetName(a, "A");
  ep = engOpen"
  if (ep .eq. 0) then
      write(6,*), "Can't start MATLAB engine."
      stop
  endif

  engPutArray(ep, a);
  engEvalString(ep, "d=eig(A*A')");
  d = engGetArray(ep, "d");
  engClose(ep);

  Dreal = mxGetPr(d);
  Dimag = mxGetPi(d);

  if(Dimag)
      printf("Eigval 2: %g+%gi\n", Dreal[1], Dimag[1]);
  else
      printf("Eigval 2: %g\n", Dreal[1]);

  mxFreeArray(a);
  mxFreeArray(d);
end
```

ActiveX Automation for Windows

MATLAB on Windows supports ActiveX Automation server capabilities. ActiveX is a component integration technology for Windows; Automation is an ActiveX protocol that allows one application or component (the “controller”) to control another application or component (the “server”). Thus, MATLAB can be launched and controlled by any Windows program that can be an Automation Controller. Some examples of applications that can be Automation Controllers are Microsoft Excel, Microsoft Access, Microsoft Project, and many Visual Basic and Visual C++ programs. Using Automation, you can execute MATLAB commands, and get and put mxArrays from and to the MATLAB workspace.

To use MATLAB as an automation server:

1. For a controller to start MATLAB, the Windows registry needs to contain certain ActiveX entries for MATLAB. To make these entries, run MATLAB from the command line as follows (you only need to do this once, unless MATLAB is moved to another location, in which case this process should be repeated):

   ```
   matlab /Regserver
   ```

   This causes MATLAB to update the registry with the appropriate entries, after which it will remain running and minimized. MATLAB can and should be shut down immediately.

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

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

```
Dim MatLab As Object
Dim Result As String
Set MatLab = CreateObject("Matlab.Application")
Result = MatLab.Execute("surf(peaks)")
```

### MATLAB ActiveX Automation Methods

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

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

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

```
Dim MatLab As Object
Dim Result As String
Set MatLab = CreateObject("Matlab.Application")
Result = MatLab.Execute("surf(peaks)")
```
void GetFullMatrix(
    [in] BSTR Name,
    [in] BSTR Workspace,
    [in, out] SAFEARRAY(double)* pr,
    [in, out] SAFEARRAY(double)* pi);

This method retrieves a full, 1- or 2-dimensional real or imaginary mxArray from the named workspace. The real and (optional) imaginary parts are retrieved into separate arrays of doubles.

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

**Workspace.** Identifies the workspace that contains the mxArray. Use the workspace name “base” to retrieve an mxArray from the default MATLAB workspace.

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

**pi.** Array of reals that is dimensioned to be the same size as the mxArray being retrieved. On return, this array will contain the imaginary values of the mxArray. If the requested mxArray is not complex, an empty array must be passed. In Visual Basic, an empty array is declared as Dim Mempty() as Double. A Visual Basic example of this method is:

```vbnet
Dim MatLab As Object
Dim Result As String
Dim MReal(1, 3) As Double
Dim MImag() As Double
Dim i, j As Integer
rem We assume that the connection to MATLAB exists.
Result = MatLab.Execute("a = [1 2 3 4; 5 6 7 8;]")
Call MatLab.GetFullMatrix("a", "base", MReal, MImag)

For i = 0 To 1
    For j = 0 To 3
        RealValue = MReal(i, j)
    Next j
Next i
```
void PutFullMatrix(
    [in] BSTR Name,
    [in] BSTR Workspace,
    [in] SAFEARRAY(double) pr,
    [in] SAFEARRAY(double) pi);

This method puts a full, 1- or 2-dimensional real or imaginary mxArray
into the named workspace. The real and (optional) imaginary parts are
passed in through separate arrays of doubles.

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

Workspace. Identifies the workspace into which the mxArray should be
placed. Use the workspace name "base" to put the mxArray into the
default MATLAB workspace.

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

pi. Array of reals that contains the imaginary values for the mxArray. If
the mxArray that is being sent is not to be complex, an empty array must
be passed for this parameter. In Visual Basic, an empty array is declared
as Dim Mempty() as Double. A Visual Basic example of this method is:

Dim MatLab As Object
Dim MReal(1, 3) As Double
Dim MImag() As Double
Dim i, j As Integer

For i = 0 To 1
    For j = 0 To 3
        MReal(i, j) = i * j;
    Next j
Next i

rem We assume that the connection to MATLAB exists.
Call MatLab.PutFullMatrix(“a”, “base”, MReal, MImag)
void MinimizeCommandWindow();

This method causes the MATLAB command window to become minimized if it is not already minimized. It accepts no arguments and has no return value.

Call Matlab.MinimizeCommandWindow

void MaximizeCommandWindow();

This method causes the MATLAB command window to restore itself to its previous size if it is not already restored. It accepts no arguments and has no return value.

Call Matlab.MaximizeCommandWindow

Additional ActiveX Information

Launching the MATLAB ActiveX Server

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

Establishing Multiple Connections

MATLAB is a “multiple use” automation server, which means that multiple controller clients can and will connect to a single instance of the MATLAB ActiveX server application. For example, suppose you have two Visual Basic programs that use MATLAB as an ActiveX automation server. When the first program is started and attempts to establish a connection to MATLAB, Windows will look to see if a MATLAB ActiveX automation server is already running. Assuming that one is not, it will automatically launch MATLAB with the /Automation command line argument. When the second program is started and a connection is established, Windows will connect the second program to the already running instance of MATLAB. This is important to note, since the two programs
will share all other resources in the MATLAB application, including variables in the MATLAB workspace.

**DCOM: Using MATLAB ActiveX over a Network Connection**

Distributed Component Object Model (DCOM) is a protocol that allows ActiveX connections to be established over a network. If you are using a version of Windows that supports DCOM (Windows NT 4.0 at the time of this writing) and a controller that supports DCOM, you can use the controller to launch MATLAB on a remote machine. To do this, DCOM must be configured properly, and MATLAB must be installed on each machine that is used as a client or server. (Even though the client machine will not be running MATLAB in such a configuration, the client machine must have a MATLAB installation because certain MATLAB components are required to establish the remote connection.) Consult the DCOM documentation for how to configure DCOM for your environment.
Dynamic Data Exchange (DDE)

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

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

- DDE concepts and terminology
- Using MATLAB as a DDE server
- Accessing other DDE servers from MATLAB
- DDE advisory links

DDE Concepts and Terminology

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

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

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

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

During a DDE conversation, the client and server applications exchange data concerning items. An item is a reference to data that is meaningful to both applications in a conversation. Either application can change the item during a conversation. These concepts are discussed in more detail below.
The Service Name
Every application that can be a DDE server has a unique service name. The service name is usually the application's executable file name without any extension. Service names are not case sensitive. Here are some commonly used service names:

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

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

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

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

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

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

- **Metafilepict**—Metafilepict format is a description of graphical data, containing the drawing commands for graphics. As a result, data stored in this format is scalable and device independent. MATLAB supports Metafilepict format for obtaining the result of a remote command that causes some graphic action to occur.

- **XLTable**—XLTable format is the clipboard format used by Microsoft Excel and is supported for ease and efficiency in exchanging data with Excel. XLTable format is a binary buffer with a header that describes the data held in the buffer. For a full description of XLTable format, consult the Microsoft Excel SDK documentation.

### Accessing MATLAB as a Server

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

- If you are using an application that provides functions or macros to conduct DDE conversations, you can use these functions or macros. For example, Microsoft Excel, Word for Windows, and Visual Basic provide DDE functions or macros. For more information about using these functions or macros, see the appropriate documentation.

- If you are creating your own application, you can use the MATLAB Engine Library or DDE directly. For more information about using the Engine Library, see “Interprocess Communication: The MATLAB Engine.” For more information about using DDE routines, see the Microsoft Windows Programmer’s Guide.

The figure below illustrates how MATLAB communicates as a server. DDE functions in the client application communicate with MATLAB’s DDE server module. The client’s DDE functions can be provided by either the application or the MATLAB Engine Library.
The DDE Name Hierarchy
When you access MATLAB as a server, you must specify its service name, topic, and item. The figure below illustrates the MATLAB DDE name hierarchy. Topics and items are described in more detail below.

MATLAB DDE Topics
MATLAB topics are System and Engine:

- The System topic allows users to browse the list of topics provided by the server, the list of System topic items provided by the server, and the formats supported by the server. These items are described in more detail below.
- The Engine topic allows users to use MATLAB as a server by passing it a command to execute, requesting data, or sending data. These items are also described in more detail below.
MATLAB System Topic Support. The MATLAB System topic supports these items:

- *SysItems* provides a tab-delimited list of items supported under the System topic (this list).
- *Format* provides a tab-delimited list of string names of all the formats supported by the server. MATLAB supports Text, Metafilepict, and XLTable. These formats are described above in the “Clipboard Formats” section.
- *Topics* provides a tab-delimited list of the names of the topics supported by MATLAB.

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

Sending Commands to MATLAB for Evaluation—Clients send commands to MATLAB using the DDE execute operation. The Engine topic supports DDE execute in two forms because some clients require that you specify the item name and the command to execute, while others require only the command. Where an item name is required, use `EngEvalString`. In both forms, the format of the command must be Text. Most clients default to Text for DDE execute. If the format cannot be specified, it is probably Text. The table shows a summary of the DDE execute parameters.

<table>
<thead>
<tr>
<th>Item</th>
<th>Format</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EngEvalString</code></td>
<td>Text</td>
<td>String</td>
</tr>
<tr>
<td><code>null</code></td>
<td>Text</td>
<td>String</td>
</tr>
</tbody>
</table>

Requesting Data from MATLAB—Clients request data from MATLAB using the DDE request operation. The Engine topic supports DDE request for three functions: requesting the text that is the result of the previous DDE execute command, requesting the graphical results of the previous DDE execute command, and requesting the data for a specified matrix.
You request the string result of a DDE execute command using the `EngStringResult` item with Text format.

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

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

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

The table shows a summary of the DDE request parameters.

<table>
<thead>
<tr>
<th>Item</th>
<th>Format</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EngStringResult</code></td>
<td>Text</td>
<td>String</td>
</tr>
<tr>
<td><code>EngFigureResult</code></td>
<td>Text</td>
<td>Yes/No</td>
</tr>
<tr>
<td><code>EngFigureResult</code></td>
<td>Metafilepict</td>
<td>Metafile of the current figure</td>
</tr>
<tr>
<td><code>&lt;matrix name&gt;</code></td>
<td>Text</td>
<td>Character buffer, tab-delimited columns, CR/LF-delimited rows</td>
</tr>
<tr>
<td><code>&lt;matrix name&gt;</code></td>
<td>XLTable</td>
<td>Binary data in a format compatible with Microsoft Excel</td>
</tr>
</tbody>
</table>

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

The table shows a summary of the DDE poke parameters.

<table>
<thead>
<tr>
<th>Item</th>
<th>Format</th>
<th>Poke Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;matrix name&gt;</td>
<td>Text</td>
<td>Character buffer, tab-delimited columns, CR/LF-delimited rows</td>
</tr>
<tr>
<td>&lt;matrix name&gt;</td>
<td>XLTable</td>
<td>Binary data in a format compatible with Microsoft Excel</td>
</tr>
</tbody>
</table>

Example: Using Visual Basic and the MATLAB DDE Server

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

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

        rem Initiate the conversation between the TextInput
        rem control and MATLAB under the Engine topic.
        rem Set the item to EngEvalString.
        TextInput.LinkMode = NONE
        TextInput.LinkTopic = "MATLAB|Engine"
        TextInput.LinkItem = "EngEvalString"
        TextInput.LinkMode = MANUAL

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

        rem Perform DDE Execute with the command string.
        TextInput.LinkExecute szCommand
        TextInput.LinkMode = NONE
```

5-20
rem Initiate the conversation between the TextOutput rem control and MATLAB under the Engine topic. rem Set the item to EngStringResult.
    TextOutput.LinkMode = NONE
    TextOutput.LinkTopic = "MATLAB|Engine"
    TextOutput.LinkItem = "EngStringResult"
    TextOutput.LinkMode = MANUAL

rem Request the string result of the previous EngEvalString rem command. The string ends up in the text field of the rem control TextOutput.text.
    TextOutput.LinkRequest
    TextOutput.LinkMode = NONE

End If
End Sub

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

This figure illustrates how MATLAB communicates as a client to a server application.
MATLAB’s DDE client module includes a set of functions. The functions that enable you to use MATLAB as a client are as follows:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddeadv</td>
<td>Set up advisory link between MATLAB and DDE server application.</td>
</tr>
<tr>
<td>ddeexec</td>
<td>Send execution string to DDE server application.</td>
</tr>
<tr>
<td>ddeinit</td>
<td>Initiate DDE conversation between MATLAB and another application.</td>
</tr>
<tr>
<td>ddepoke</td>
<td>Send data from MATLAB to DDE server application.</td>
</tr>
<tr>
<td>ddereq</td>
<td>Request data from DDE server application.</td>
</tr>
<tr>
<td>ddeterm</td>
<td>Terminate DDE conversation between MATLAB and server application.</td>
</tr>
<tr>
<td>ddeunadv</td>
<td>Release advisory link between MATLAB and DDE server application.</td>
</tr>
</tbody>
</table>

If the server application is Microsoft Excel, you can specify the System topic or a topic that is a filename. If you specify the latter, the filename ends in .XLS or .XLC and includes the full path if necessary. A Microsoft Excel item is a cell reference, which can be an individual cell or a range of cells.

Microsoft Word for Windows topics are System and document names that are stored in files whose names end in .DOC or .DOT. A Word for Windows item is any bookmark in the document specified by the topic.
The following example is an M-file that establishes a DDE conversation with Microsoft Excel, then passes a 20-by-20 matrix of data to Excel.

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

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

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

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

**DDE Advisory Links**

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

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

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

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

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

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

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

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

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

% Poke the z data to the Excel spreadsheet.
r = ddepoke(chan, range, z);

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

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

6-2 Configuration
6-2 Troubleshooting MEX-File Creation
6-4 Understanding MEX-File Problems

6-7 Custom Building of MEX-Files
6-9 UNIX
6-11 Windows
6-14 Macintosh—MPW
6-15 Macintosh—Metrowerks CodeWarrior
Configuration

Your installed version of MATLAB contains all the tools you need to use the Application Program Interface, except a C or Fortran compiler. Depending on your requirements, you’ll need either an ANSI C compiler or a Fortran 77 compiler. Also, if you are working on a Windows platform, your compiler must be able to create 32-bit Windows dynamically linked libraries (DLLs).

In certain cases you may need to know more detailed information about how the build scripts work. The section, “Custom Building of MEX-Files,” describes the scripts and how they work on the Windows, UNIX, and Macintosh platforms.

Troubleshooting MEX-File Creation

Use this flow chart to help isolate difficulties in creating MEX-files. The section, “Understanding MEX-File Problems,” following the flow chart provides additional information regarding common problems that occur when creating MEX-files. If the suggestions in these sections do not help, access the Solutions Search Engine at http://www.mathworks.com/solution.html.
Start: Can you compile and run timestwo.c or timestwo.f?

1. Are you using a supported compiler? 
   - no: Acquire a supported compiler. See “System Setup” for details.
   - yes: Double check your configuration; see “System Setup”.

2. Can you compile your program?
   - no: Check for:
     - ANSI C code
     - General C syntax errors
   - yes: Check:
     - Spelling of mexFunction
     - Link against all libraries you intend to use.

3. Can MATLAB load your MEX-file?
   - no: Use:
     - mexprint
     - mex -argcheck
   - yes: Use:
     - matplot -check_malloc

4. Segmentation fault or bus error?
   - no: Do you get the right answer?
     - no: Use:
       - matplot -check_malloc
       - Run in debugger.
     - yes: Stop
   - yes: Use:
     - matplot -check_malloc
     - Run in debugger.

5. Stop

---

1 UNIX only
2 MEX-files only
Understanding MEX-File Problems

Problem 1
The most common configuration problem in creating C MEX-files on UNIX involves using a non-ANSI C compiler, or failing to pass to the compiler a flag that tells it to compile ANSI C code.

A reliable way of knowing if you have this type of configuration problem is if the header files supplied by The MathWorks generate a string of syntax errors when you try to compile your code. See “Specifying an Options File” in Chapter 1 for information on selecting the appropriate options file or, if necessary, obtain an ANSI C compiler.

Problem 2
A second way of generating a string of syntax errors occurs when you attempt to mix ANSI and non-ANSI C code. The MathWorks provides header and source files that are ANSI C compliant. Therefore, your C code must also be ANSI compliant.

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

Problem 3
If you receive an error of the form:

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

MATLAB is unable to recognize your MEX-file as being valid.

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

On some platforms, if you fail to link against required libraries, you may get an error when MATLAB loads your MEX-file rather than when you compile your MEX-file. In such cases, you see a system error message referring to “unresolved symbols” or “unresolved references.” Be sure to link against the library that defines the function in question.
On Windows, MATLAB will fail to load MEX-files if it cannot find all DLLs referenced by the MEX-file; the DLLs must be on the path or in the same directory as the MEX-file. This is also true for third party DLLs.

Problem 4

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

Segmentation violations do not always occur at the same point as the logical errors that cause them. If a program writes data to an unintended section of memory, an error may not occur until the program reads and interprets the corrupted data. Consequently, a segmentation violation or bus error can occur after the MEX-file finishes executing.

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

- Recompile your MEX-file with argument checking (C MEX-files only). You can add a layer of error checking to your MEX-file by recompiling with the `mex` script flag `-argcheck`. This warns you about invalid arguments to both MATLAB MEX-file (mex) and matrix access (mx) API functions.
  
  Although your MEX-file will not run as efficiently as it can, this switch prevents 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 prevents such errors as writing past the end of a dimensioned array, or freeing previously freed memory.
• Run MATLAB within a debugging environment. This process is already described in the chapters on creating C and Fortran MEX-files, respectively.

Problem 5
If your program generates the wrong answer(s), there are several possible causes. First, there could be an error in the computational logic. Second, the program could be reading from an uninitialized section of memory. For example, reading the 11th element of a 10-element vector yields unpredictable results.

Another possibility for generating a wrong answer could be overwriting valid data due to memory mishandling. For example, writing to the 15th element of a 10-element vector might overwrite data in the adjacent variable in memory. This case can be handled in a similar manner as segmentation violations as described in Problem 4.

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

This section discusses in detail the process that the MEX-file build script uses. In general, the defaults that come with MATLAB should be sufficient for building most MEX-files. There are reasons that you might need more detailed information, such as:

- You want to use an Integrated Development Environment (IDE), rather than the provided script, to build MEX-files.
- You want to create a new options file, for example to use a compiler that is not directly supported.
- You want to exercise more control over the build process than the script uses.

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

The `mex` script has a set of switches that you can use to modify the link and compile stages. This table lists the available switches and their uses.

<table>
<thead>
<tr>
<th>Switch</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-argcheck</td>
<td>Perform argument checking on MATLAB API functions.</td>
</tr>
<tr>
<td>-c</td>
<td>Compile only; do not link.</td>
</tr>
<tr>
<td>-D&lt;name&gt;</td>
<td>Define preprocessor symbol <code>&lt;name&gt;</code>.</td>
</tr>
<tr>
<td>-f &lt;file&gt;</td>
<td>Use <code>&lt;file&gt;</code> as the options file. Note that under Windows, you must specify the full path to the options file since the <code>PATH</code> environment variable is not used to locate the file.</td>
</tr>
<tr>
<td>-g</td>
<td>Debug; i.e., use the <code>COMPDEBUGFLAGS</code> and <code>LINKDEBUGFLAGS</code> (see the options file for additional information).</td>
</tr>
<tr>
<td>-help</td>
<td>Help; lists the switches and their functions.</td>
</tr>
</tbody>
</table>
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.

For UNIX, the default options file provided with MATLAB is located in $MATLAB/bin. For Windows, the default is in matlab\bin. For Macintosh, the default options file is in the $MATLAB:extern:scripts: folder.

The mex script will look for an options file called mexopts.sh (UNIX) or mexopts.bat (Windows) first in the current directory. Then, depending on your platform, it continues to search for the options file. On UNIX systems, it searches next in your $HOME/matlab directory, and finally in $MATLAB/bin. On Windows, it searches next in matlab\bin; it does not search the path. On Macintosh, it searches next in the $MATLAB:extern:scripts: folder. You can also directly specify the name of the options file using the -f switch.

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file in $MATLAB/bin/mexopts.sh (matlab\bin\mexopts.bat in Windows), or you can invoke the mex script in verbose mode.
The following section provides more detail regarding each of these stages. However, there is a general way to obtain specifics on the build process, which is the verbose option to the `mex` script (the `-v` flag). This will print the exact compiler options, prelink commands (if appropriate), and linker options used. The following section gives an overview of the high-level process; for exact flags provided for each compiler, invoke the `mex` script with the verbose flag.

**UNIX**

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

- Add `$MATLAB/extern/include` to the list of directories in which to find header files (`-I$MATLAB/extern/include`)

- Define the preprocessor macro `MATLAB_MEX_FILE` (`-DMATLAB_MEX_FILE`)

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

For all platforms except SunOS 4.x, the link stage must:

- Instruct the linker to build a shared library
- Link all objects from compiled source files (including `mexversion.c`)
- (Fortran MEX-files only) Link in the precompiled versioning source file, `$MATLAB/extern/lib/$Arch/version4.o`
- Export the symbols `mexFunction` and `mexVersion` (these symbols represent functions called by MATLAB)

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

On the SunOS 4.x platform, the link stage is more complicated. The `mex` script does a “dryrun” of the linker to see what libraries need to be linked in, and what flags need to be used. The output of the dryrun, and the final flags and libraries used, are different for each compiler and compiler version. However, they are displayed in the verbose output.
For customizing the build process, you should modify the options file. The options file contains the compiler-specific flags corresponding to the general steps outlined above. The options file consists of a series of variable assignments; each variable represents a different logical piece of the build process. The options files provided with MATLAB are located in $\text{MATLAB/bin}$.

The `mex` script looks for an options file called `mexopts.sh` first in the current directory, then in your $\text{HOME/matlab}$ directory, and finally in $\text{MATLAB/bin}$. You can also directly specify the name of the options file using the `-f` option.

To aid in providing flexibility, there are two sets of options in the options file that can be turned on and off with switches to the `mex` script. These sets of options correspond to building in “debug mode” and building in “optimization mode.” They are represented by the variables `DEBUGFLAGS` and `OPTIMFLAGS`, respectively, one pair for each “driver” that is invoked (C`DEBUGFLAGS` for the C compiler, F`DEBUGFLAGS` for the Fortran compiler, and L`DEBUGFLAGS` 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:
### Custom Building of MEX-Files

#### Windows

There are three stages to MEX-file building for both C and Fortran on Windows—compiling, prelinking, and linking. The compile stage must:

1. 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.
2. Define the name of the compiler, using the `COMPILER` environment variable, if needed.
3. Define the compiler switches in the `COMPLFLAGS` environment variable.
   1. The switch to create a DLL is required for MEX-files.
   2. For stand-alone programs, the switch to create an `exe` is required.
   3. The `-c` switch (compile only; do not link) is recommended.
   4. Any other switch specific to the environment can be used.
4. Define preprocessor macro, with `-D`, `MATLAB_MEX_FILE` is required.
5. Set up optimizer switches and/or debug switches using `OPTIMFLAGS` and `DEBUGFLAGS`. These are mutually exclusive: the `OPTIMFLAGS` are the

#### Table: MEX-File Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>C Compiler</th>
<th>Fortran Compiler</th>
<th>Linker</th>
</tr>
</thead>
<tbody>
<tr>
<td>executable</td>
<td>CC</td>
<td>FC</td>
<td>LD</td>
</tr>
<tr>
<td>flags</td>
<td>CFLAGS</td>
<td>FFLAGS</td>
<td>LDFLAGS</td>
</tr>
<tr>
<td>optimization</td>
<td>COPTIMFLGS</td>
<td>FOPTIMFLGS</td>
<td>LDOPTIMFLGS</td>
</tr>
<tr>
<td>debugging</td>
<td>CDEBUGFLGS</td>
<td>FDEBUGFLGS</td>
<td>LDDEBUGFLGS</td>
</tr>
<tr>
<td>additional libraries</td>
<td>CLIBS</td>
<td>FLIBS</td>
<td>----</td>
</tr>
</tbody>
</table>

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.
default, and the DEBUGFLAGS are used if you set the -g switch on the mex command line.

The prelink stage dynamically creates import libraries to import the required function into the MEX, MAT, or engine file. All MEX-files link against MATLAB only. MAT stand-alone programs link against libmx.dll (array access library) and libmat.dll (MAT-functions). Engine stand-alone programs link against libmx.dll (array access library) and libeng.dll for engine functions. Each DLL and MATLAB has a corresponding .def file of the same name located in the matlab\extern\include directory.

Finally, the link stage must:

- Define the name of the linker in the LINKER environment variable.
- Define the LINKFLAGS environment variable that must contain:
  - The switch to create a DLL for MEX-files, or the switch to create an exe for stand-alone programs.
  - Export of the entry point to the MEX-file as mexFunction for C or MEXFUNCTION@16 for Microsoft Fortran.
  - The import library(s) created in the PRELINK_CMDS stage.
  - Any other link switch specific to the compiler that can be used.
- Define the linking optimization switches and debugging switches in LINKEROPTIMFLAGS and LINKDEBUGFLAGS. As in the compile stage, these two are mutually exclusive: the default is optimization, and the -g switch invokes the debug switches.
- Define the link-file identifier in the LINK_FILE environment variable, if needed. For example, Watcom uses file to identify that the name following is a file and not a command.
- Define the link-library identifier in the LINK_LIB environment variable, if needed. For example, Watcom uses library to identify the name following is a library and not a command.
- Optionally, set up an output identifier and name with the output switch in the NAME_OUTPUT environment variable. The environment variable MEX_NAME contains the name of the first program in the command line. This must be set for -output to work. If this environment is not set, the compiler default is to use the name of the first program.
in the command line. Even if this is set, it can be overridden by specifying the \texttt{mex -output} switch.

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

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

1. Create a project and insert your MEX source into it.
2. Create a \texttt{.DEF} file to export the MEX entry point. For example:
   \begin{verbatim}
   LIBRARY MYFILE.DLL
   EXPORTS mexFunction <- for a C MEX-file
   or
   EXPORTS MEXFUNCTION@16 <- for a Fortran MEX-file
   \end{verbatim}
3. Add the \texttt{.DEF} file to the project.
4. Create an import library of MEX-functions from \texttt{MATLAB.DEF} using the \texttt{LIB} command. For example:
   \begin{verbatim}
   LIB /DEF:MATLAB\EXTERN\INCLUDE\MATLAB.DEF /OUT:mymeximports.lib
   \end{verbatim}
5. Add the import library to the library modules in the \texttt{LINK settings} option.
6. Add the \texttt{MATLAB\include} directory, \texttt{MATLAB\EXTERN\INCLUDE} to the \texttt{include} path in the \texttt{settings C/C++ preprocessor} option.
7. Add \texttt{MATLAB\_MEX\_FILE} to the \texttt{settings C/C++ preprocessor} option.
8. To debug the MEX-file using the IDE, put \texttt{MATLAB.EXE} in the \texttt{settings debug} option as the \texttt{executable for debug session}. 


Macintosh—MPW

On Macintosh and Power Macintosh systems using the MPW compiler, there are three stages in MEX-file building: compiling, linking, and resource compiling. The compiler stage must:

- Add \texttt{\$MATLAB:extern:include} to the list of directories in which header files reside (\texttt{-i \$MATLAB:extern:include})
- Define the \texttt{MATLAB\_MEX\_FILE} preprocessor symbol (\texttt{-d MATLAB\_MEX\_FILE})
- Specify that enumerated types are size \texttt{int} (\texttt{-enum int})

The link stage for Power Macintoshes must:

- Specify \texttt{mexFunction} as an export function and as the main entry point (\texttt{-main mexFunction -export mexFunction})
- Specify the type (\texttt{MEX0}) and creator (\texttt{MATL}) of the MEX-file (\texttt{-t MEX0 -c MATL})
- Link against the MATLAB binary and the compiler-provided libraries needed to define any ANSI or Macintosh OS functions called by the MEX-file.

The link stage for 68K Macintosh must:

- Specify \texttt{mexmain} as the main entry point (\texttt{-m mexmain})
- Specify the code resource type and id for a MEX-file (\texttt{-rt MEX0=0 -sg MEX})
- Specify the type (\texttt{MEX0}) and creator (\texttt{MATL}) of the MEX-file (\texttt{-t MEX0 -c MATL})
- Link against \texttt{libmex.o} and the compiler-provided libraries needed to define any ANSI or Macintosh OS function called by the MEX-file

The resource compile stage must:

- Specify \texttt{mxVR.r} as a Rez source file
- Specify the \texttt{V4\_COMPAT} and \texttt{ARRAY\_ACCESS\_INLINE\_NG} macros if these were also passed to the source compiler (\texttt{-d V4\_COMPAT -d ARRAY\_ACCESS\_INLINE\_NG})
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 default options files provided with MATLAB for MPW use are `mexopts.mpwc` (C) and `mexopts.ls` (Fortran) and are in the `$MATLAB:extern:scripts:` folder.

Options files for use with MPW compilers are written as MPW scripts, while options files for use with Metrowerks’ compilers (i.e., `mexopts.cw`) are written as M-file scripts. To distinguish between these two languages, MPW `mexopts` files must start with the line:

```
#MPW
```

The `mex` script looks for an options file called `mexopts` first in the current directory and then in the `$MATLAB:extern:scripts:` folder. You can also directly specify the name of the options file using the `-f` options. 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 “optimization mode.” These options are represented by `DEBUGFLAGS/LINKDEBUGFLAGS` and `OPTIMFLAGS/LINKOPTIMFLAGS`, respectively. Depending on how you decide to build your MEX-file, the `mex` script will include one or both sets of options:

- If you build in optimization mode (the default), the `mex` script will include the `OPTIMFLAGS` options in the compile stage and the `LINKOPTIMFLAGS` in the link stage.
- If you build in debug mode, the `mex` script will include the `DEBUGFLAGS` options in the compile stage and the `LINKDEBUGFLAGS` in the link stage.
- If you specify both optimization and debug modes, both sets of options will be specified in the compile and link stages.

**Macintosh—Metrowerks CodeWarrior**

On Macintosh 68K and Power Macintosh systems using the CodeWarrior compiler, you must first set up a CodeWarrior project, from which you can build the MEX-file. You can use the `mex.m` M-file to create a CodeWarrior project and automatically build the MEX-file from the
project. MATLAB communicates with CodeWarrior via AppleScript technology.

For customizing the build process, you 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 default options file provided with MATLAB for Metrowerks use is `mexopts.cw` and is in the `$MATLAB:extern:scripts:` folder.

Unlike MPW compilers, which accept options files written as MPW scripts, Metrowerks' compilers accept options files written as M-files. To distinguish between these two languages, CodeWarrior `mexopts` files must start with the line:

```
%METROWERKS
```

The `mex.m` script looks for an options file called `mexopts` first in the current directory and then in `$MATLAB:extern:scripts:` folder. You can also directly specify the name of the options file using the `-f` option. To provide more flexibility, there are two sets of options in the options file that you can turn on or off with switches to the `mex` script. These sets of options correspond to build in “debug” and “optimization” modes, are represented by the variables `DEBUGFLAGS/LINKDEBUGFLAGS` and `OPTIMFLAGS/LINKOPTIMFLAGS`, respectively. Depending on how you decide to build your MEX-file, the `mex` script will include one or both sets of options:

- If you build in optimization mode (the default), the `mex` script will include the `OPTIMFLAGS` options in the compile stage and the `LINKOPTIMFLAGS` in the link stage.
- If you build in debug mode, the `mex` script will include the `DEBUGFLAGS` options in the compile stage and the `LINKDEBUGFLAGS` in the link stage.
- If you specify both optimization and debug modes, both sets of options will be specified in the compile and link stages.
The four **FLAGS** variables are specified in AppleScript syntax because `mex.m` communicates with CodeWarrior via AppleScript. This table lists the **FLAGS** variables and the preference panel they control:

<table>
<thead>
<tr>
<th><strong>FLAG</strong> Variable</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COMPFLAGS</strong></td>
<td>Preferences for CodeWarrior C’s “C/C++ Language” preference panel (also called the “C/C++ Compiler” panel).</td>
</tr>
<tr>
<td><strong>LINKFLAGS</strong></td>
<td>Preferences for CodeWarrior C’s “PPC Linker” or “68K Linker” preference panel.</td>
</tr>
<tr>
<td><strong>OPTIMFLAGS</strong></td>
<td>Preferences for CodeWarrior C’s “PPC Processor” preference panel (also called the “PPC CodeGen” panel) or “68K Processor” preference panel (also called the “68K CodeGen” panel).</td>
</tr>
<tr>
<td><strong>DEBUGFLAGS</strong></td>
<td>Preferences for CodeWarrior C’s “PPC Linker” or “68K Linker” preference panel. This is the same as the <strong>LINKFLAGS</strong> variable because both linking and debugging preferences are set in this panel.</td>
</tr>
</tbody>
</table>

The `mex` script first wraps the values of these variables in curly braces ("{ " and "}") to make them into AppleScript lists, and then uses these lists to set various preference panel values. You can find a description of the names and legal values in the “CodeWarrior Apple Events” chapter of the CodeWarrior User’s Guide.

For example, if you want to optimize a Power Macintosh MEX-file for the PowerPC 604 processor, the following AppleScript code sets the **Instruction Scheduling** optimization pop-up menu to “604” in the **PPC Processor** preference panel:

```appleScript
set Preferences of panel 'PPC CodeGen' to ¬
{Instruction Scheduling: PowerPC604}
```

(“¬” is the AppleScript line continuation character.)

The **PPC CodeGen** preference panel corresponds to the `mexopts` variable **OPTIMFLAGS**. This means that you should set the value of
OPTIMFLAGS in your mexopts file to the string inside the curly brackets in the AppleScript command shown above:

```
OPTIMFLAGS='Instruction Scheduling: PowerPC604'
```

After `mex.m` creates a CodeWarrior project for your MEX-file, you can use that project file to rebuild the MEX-file (for example, if you need to make changes to the source code of the MEX-file).

Alternatively, you can follow these instructions to set up a MEX-file CodeWarrior project without using `mex.m`.

**CodeWarrior: Building on a Power Macintosh**

To build MEX-files on a Power Macintosh, you must perform these steps:

1. Select **File...New Project** to create a new project file. In the dialog box, enter the project name (e.g., `explore`). Then, set the **Project Stationery** pop-up to **–ANSI PPC C/C++**, and click **Save**. A starting list of files appears.

2. From the file list, select the files
   - `<replace me ANSI>.c`
   - `ANSI C++.PPC.Lib`
   - `SIOUX.PPC.Lib`

   Select **Project...Remove Files** to delete the selected files from the project.
3 Select **Project...Add Files** and add the files

- MATLAB:extern:include:mxVR.r for regular MEX-files
- MATLAB:MATLAB, the MATLAB executable (In the project file list, this must precede all CodeWarrior libraries.)
- All source code, for example, *timestwo.c*
- Any third party libraries that the MEX application requires, for example, GUI libraries

A message window appears showing the addition of access paths.

**Note:** MATLAB (in italics) means the top-level folder in which MATLAB has been installed.

4 Select the **Edit...Preferences** menu item. Steps 5 through 12 take place in submenus of **Preferences**, and they assume that the starting conditions are set to **Factory Settings**. Steps 5 through 12 describe the minimum set of things you must do to generate MEX-files. If you are an experienced CodeWarrior programmer, you may want to select other options as desired.

5 Select the **Target** icon. Select **Filetype APPL**. Set the **Compiler:** pop-up menu to **PEF Importer**. Click the **Change** button.

6 Select the **C/C++ Language** icon. In the **Prefix File** field, enter `CodeWarrior.MEX.prefix`. If you are building a MATLAB 4 MEX-file, duplicate the `CodeWarrior.MEX.prefix` file and place it in the `MATLAB:extern:include:` folder. Rename the file to `CodeWarrior.MEX.prefix.V4`, then edit the file by adding the line

```c
#define V4_COMPAT
```
and enter CodeWarrior.MEX.prefix.V4 in the Prefix File field. In addition, check the following check boxes:

- Enums Always Int
- MPW Newlines
- Require Function Prototypes

7 Select the PPC Processor icon. Choose any desired optimizations.

8 Select the PPC Linker icon. In the Main field, enter mexFunction.

9 Select the PPC PEF icon. From the Export Symbols pop-up menu, select Use #pragma. Also, enter mexFunction in the Fragment Name field.

10 Select the PPC Project icon. Set the Project Type pop-up menu to Shared Library. Also, in the File Name field, enter the name of the MEX-file. The name you specify must end with the letters .mex; for example, CWexample.mex is a legal name but CWexample is not. Set the Creator field to MATL. Finally, set the Type field to MEX0 (M-E-X-zero).

11 Select the Rez icon. In the Prefix File field, enter the same value entered for the Prefix File field in Step 6.

12 Click OK to lock in the changes made in Steps 5 through 11.

13 Select Project...Make to compile and link the MEX-function. You can safely ignore the warning message:

```
Link Warning: ignored: "q.d" (data) in Startup.c
Previously defined in MATLAB.xcoff
```

If there are no errors, CodeWarrior builds the target MEX-file. Invoke this MEX-file from MATLAB 5 running on a Power Macintosh.
Custom Building of MEX-Files

CodeWarrior: Building on a Motorola 680x0-based Macintosh

To build MEX-files on a Motorola 680x0-based Macintosh, you must perform these steps:

1. Create a new project file by holding down the **option** key and selecting the **File...New Project** menu item. Give the project a name, such as **timestwo**. Leave the **Project Stationery** pop-up menu set to **None**. Click **Save** and a starting list of files appears.

2. Select the **Project...Add Files** menu item. Then, add the files:
   - MATLAB:extern:include:mxVR.r for regular MEX-files.
   - MATLAB:extern:lib:68k:Metrowerks:libmex.lib (In the project file list, this library must precede all CodeWarrior libraries.)
   - CodeWarrior:Metrowerks CodeWarrior:
     ANSI (4i/F/8d):ANSI (N/4i/F/8d) C.A4.68K.Lib
   - CodeWarrior:Metrowerks CodeWarrior:
     Mac OS Support:Libraries:MacOS 68K:MathLib68K:
   - CodeWarrior:Metrowerks CodeWarrior:
     Mac OS Support:Libraries:MacOS 68K:MacOS.lib
   - All source code, for example, **timestwo.c**
   - Any third party libraries that the MEX application requires, for example, GUI libraries

**Note:** MATLAB (in italics) means the top-level folder in which MATLAB has been installed. CodeWarrior (in italics) is the top-level folder in which the CodeWarrior application has been installed.

3. Select the **Edit...Preferences** menu item. Steps 4 through 9 take place in submenus of Preferences, and assume that the starting conditions are set to **Factory Settings**. Steps 4 through 9 describe the minimum set of things you must do to generate MEX-files. If you are
an experienced CodeWarrior programmer, you may want to select other options as desired.

4 Select the **68K Processor** icon. Enter Large in the **Code Model** field. In addition, check the following check boxes:

- 68020 Codegen
- 68881 Codegen
- 4-Byte Ints
- 8-Byte Doubles

5 Select the **68K Linker** icon. Put a check in the check box

- Link Single Segment

6 Select the **68K Project** icon. Then

- Set **Project Type** to Code Resource
- Set **File Name** to the filename of the target MEX-file (for example, timestwo.mex). The target filename must end with .mex.
- In the **Creator** field, type MATL.
- In the **Type** field, type MEX0 (M-E-X-zero).
- In the **ResType** field, type MEX0 (M-E-X-zero).
- Place a check in the **Extended Resource** check box.
7 Select the **C/C++ Language** icon. In the **Prefix File** field, enter `CodeWarrior.MEX.prefix`. If you are building a MATLAB 4 MEX-file, duplicate the `CodeWarrior.MEX.prefix` file and place it in the `MATLAB:extern:include:` folder. Rename the file to `CodeWarrior.MEX.prefix.V4`, then edit the file by adding the line

```c
#define V4_COMPAT
```

and enter `CodeWarrior.MEX.prefix.V4` in the **Prefix File** field. In addition

- Place a check in the **MPW Newlines** check box.
- Place a check in the **Enums Always Int** check box.

8 Select the **Rez** icon. In the **Prefix File** field, enter the same value entered for the **Prefix File** field in Step 7.

9 Click **OK** to lock in the changes made in Steps 4 through 8.

10 Select **Project...Make** to compile and link the MEX-function. You can safely ignore the warning messages:

```
Link Warning : ignored: 'calloc' in calloc.c
Previously defined in mexcbk_mac.c
Link Warning : ignored: 'free' in free.c
Previously defined in mexcbk_mac.c
Link Warning : ignored: 'malloc' in malloc.c
Previously defined in mexcbk_mac.c
Link Warning : ignored: 'realloc' in realloc.c
Previously defined in mexcbk_mac.c
```

If there are no errors, CodeWarrior builds the target MEX-file. Invoke this MEX-file from MATLAB 5 running on a Motorola 680x0-based Macintosh.
CodeWarrior: Building FAT
To build a MEX-file that can run on either a Power Macintosh or a Motorola 680x0-based Macintosh platform, you must:

1. Create a project file for the Motorola 680x0 platform as described in Steps 1 through 9 of the “CodeWarrior: Building on a Motorola 680x0-based Macintosh” section. Do not invoke Project…Make yet.

2. In the project file for the Motorola 680x0 platform, select the Edit…Preferences menu. Then, select the 68K Project icon, and
   • In the Type field, change MEX0 to src
   • In the File Name field, modify the target filename by appending .68k to it; for example, if the target filename is timestwo.mex, change it to timestwo.mex.68k

3. In the project file for the Motorola 680x0 platform, choose Project…Make to build the target MEX-file. If there are no errors, CodeWarrior generates the target MEX-file (for example, timestwo.mex.68k).

4. Create a project file for the Power Macintosh as described in Steps 1 through 13 of the “CodeWarrior: Building on a Power Macintosh” section in this chapter. Do not invoke Project…Make yet.

5. In the project file for the Power Macintosh, select the Project…Add menu item. Add the filename of the Motorola 680x0 target MEX-file (for example, timestwo.mex.68k) to the list of files.

6. In the project file for the Power Macintosh, choose Project…Make to build the target FAT MEX-file. You can safely ignore these warning messages:

   Link Warning: ignored: 'qd' (data) in Startup.c
   Previously defined in MATLAB
   Link Warning: ignored: duplicate resource 'mxVR' (0) in 'timestwo.mex.68k'

If there are no errors, CodeWarrior builds the target MEX-file. Invoke this MEX-file from MATLAB 5 running on a Power Macintosh or on a Motorola 680x0-based Macintosh.
Symbols
%val 3-5, 3-7
  allocating memory 3-22

A
access methods 3-7
  mat 4-2
  mex 1-7
  mex 1-7
ActiveX 5-3, 5-8
  controller 5-8
  launching server 5-12
  methods 5-8
  multiple connections 5-12
  server 5-8
  using over a network 5-13
API
  access methods 1-7
  documentation 1-3
  supported functionality 1-2
array
  MATLAB 1-5
  sparse 2-28
automation
  controller 5-8
  server 5-8
  multiple use 5-12

B
BSTR 5-9

C
C language
  debugging 2-33

MEX-files 2-2
C language example
  basic 2-8
  calling MATLAB functions 2-30
  calling user-defined functions 2-30
  handling 8-, 16-, 32-bit data 2-25
  handling arrays 2-27
  handling complex data 2-23
  handling sparse arrays 2-28
  manipulating cell arrays 2-20
  manipulating structures 2-16
  passing multiple values 2-13
  strings 2-11
  cell array 2-20, 2-23
CodeWarrior
  access path 1-13
  debugging
    C 2-35
    Fortran 3-35
  special considerations 1-11
  updating 1-12
commands See individual commands.
compiler
  debugging
    CodeWarrior 2-35, 3-35
    Microsoft 2-34
    Watcom 2-34
  preconfigured options file 1-14–1-15
compiling
  engine program
    UNIX 5-5–5-6
    Windows 5-5
MAT-file
  Macintosh 4-16
  UNIX 4-15
  Windows 4-15
complex data
  in Fortran 3-20
computational routine 2-2, 3-3
configuration 1-9
  Macintosh 1-10
problems 6-4
testing 1-9
troubleshooting 1-16
UNIX 1-10
Windows 1-9
conversation 5-14

D
data
  MATLAB
    exporting from 4-3–4-4
    importing to 4-2–4-3
data structure
  mxArray 1-6
data type 2-7
  C 1-7
    checking 2-9
  Fortran 1-7, 3-2
  MATLAB 4-7
DCOM 5-13
DDE 5-14
  accessing MATLAB as server 5-16
  advisory links 5-23
  client 5-14
  conversation 5-14
  hot link 5-23
  item 5-15
MATLAB
  requesting data from 5-18
  sending commands to 5-18
  sending data to 5-19
  using as client 5-21
name hierarchy 5-17
notifying when data changes 5-23
server 5-14
service name 5-15
topic 5-15, 5-17–5-21
  engine 5-17
  system 5-17
warm link 5-23
  Windows clipboard formats 5-15–5-16
ddeadv 5-22
ddeexec 5-22
ddeinit 5-22
ddepoke 5-22
ddereq 5-22
ddeterm 5-22
ddeunadv 5-22
debugging C language MEX-files 2-33
  Macintosh 2-35
  UNIX 2-33
  Windows 2-34
debugging Fortran language MEX-files
  Macintosh 3-35
  UNIX 3-33
  Windows 3-34
diary file 4-3
directory organization
  C language MEX-file 2-2
  Fortran language MEX-file 3-2
  MATLAB 4-7
  options file 6-8
documenting MEX-file 2-31, 3-32
dynamic data exchange See DDE.
dynamic memory allocation
  in Fortran 3-22
  Microsoft Fortran 3-24
  mxMalloc 2-11
  dynamically linked subroutine 1-4

E
  engClose 5-3
  engEvalString 5-3
  engGetArray 5-3
  engine
    compiling 5-5
    linking 5-5
  engine example
    calling MATLAB
      from C program 5-3
      from Fortran program 5-7
  engine library 5-2
    communicating with MATLAB
      UNIX 5-3
      Windows 5-3
  engOpen 5-3
  engOutputBuffer 5-3
  engPutArray 5-3

F
  file mode
    ASCII 4-5
    binary 4-5
  files
    linking multiple 2-32, 3-32
  Fortran
    %val 3-5
    case in 3-6
  pointers
    concept 3-5, 3-14
    declaring 3-5
    differing sizes 3-5
  Fortran language example
    calling MATLAB functions 3-28
    handling complex data 3-20
    handling sparse matrices 3-26
    passing arrays of strings 3-13
    passing matrices 3-14
    passing multiple values 3-17
    passing scalar 3-9
    passing strings 3-11
  Fortran language MEX-files 3-2, 3-3
    components 3-3
    passing integer 3-28

G
  gateway routine 2-2, 3-3, 3-6
  accessing mxArray data 2-3
  GetFullMatrix 5-10

H
  help 2-31, 3-32
  Help Desk 1-3
  help files 2-31

I
  ir 1-6, 2-28, 3-26
J
Jasik Debugger
  debugging C
    on 68K 2-36
    on Power Mac 2-38
  debugging Fortran
    on 68K 3-35
    on Power Mac 3-37
jc 1-6, 2-28, 3-26

L
Language Systems (LS) Fortran
  special considerations 1-13
  linking multiple files 2-32, 3-32
load 4-5

M
Macintosh
  special considerations 1-11–1-13
matClose 4-5
matDeleteArray 4-6
matDeleteMatrix 4-6
MAT-file
  C language
    creating 4-11
    reading 4-13
  compiling 4-15
  data types 4-5
  examples 4-10
  Fortran language
    creating 4-12
    reading 4-14
    linking 4-15

LS Fortran
  68K 4-20
  Power Mac 4-19
  Macintosh 4-16–4-21
  Metrowerks C
    68K 4-17
    Power Mac 4-16
  MPW C
    68K 4-18
    Power Mac 4-17
  subroutines 4-5
  UNIX libraries 4-9
  using 4-2
  Windows libraries 4-9
MAT-functions 4-5–4-7
  matGetArray 4-6
  matGetArrayHeader 4-7
  matGetDir 4-5
  matGetFp 4-6
  matGetFull 4-6
  matGetMatrix 4-6
  matGetNextArray 4-6
  matGetNextArrayHeader 4-7
  matGetNextMatrix 4-6
  matGetString 4-6
MATLAB
  array 1-5
  data file format 4-2
  data types 1-7
  engine 5-2
  exporting data 4-2–4-4
  importing data 4-2–4-3
  MAT-file 4-5
    reading arrays from 4-5
    saving arrays to 4-5
    moving data between platforms 4-4–4-5
stand-alone applications 4-2
starting from controller 5-8
using as a computation engine 5-2
variables 1-5
matOpen 4-5
matPutArray 4-6
matPutArrayAsGlobal 4-7
matPutFull 4-6
matPutMatrix 4-6
matPutString 4-6
matrix
  full 1-6
  sparse 1-6, 3-26
MaximizeCommandWindow 5-12
memory allocation 2-11
method
  ActiveX 5-8
  BSTR 5-9
  Execute 5-8
  GetFullMatrix 5-10
  MaximizeCommandWindow 5-12
  MinimizeCommandWindow 5-12
  PutFullMatrix 5-11
mex script 6-7
  searching for options file 6-8
  switches 6-7
mex.m 2-9
mexCallMATLAB 2-30, 3-28
mexErrMsgTxt 3-7
MEX-file 1-4
  arguments 2-5
  C language 2-2
  calling 1-5
compiling 2-9
  CodeWarrior 6-15–6-24
  Macintosh 1-10
  Microsoft Visual C++ 6-13
  MPW 6-14–6-15
  UNIX 1-10, 6-9–6-11
  Windows 1-9, 6-11–6-13
components 2-2
  computation error 6-6
  configuration problem 6-4
  creating C language 2-2, 2-9
  creating Fortran language 3-2
  custom building 6-7
  data structures 3-7
  debugging C language 2-33
  debugging Fortran language 3-33
  documenting 2-31, 3-32
  extensions 1-5
  load error 6-4
  passing structures 2-15
  problems 6-4–6-6
  segmentation error 6-5
  syntax errors 6-4
  use of 1-4
mexFunction 2-2, 3-3, 3-6
  parameters 2-2
mexMakeMemoryPersistent 2-11
mexopts file 1-10
M-file 4-3
Microsoft compiler
  debugging 2-34
  MinimizeCommandWindow 5-12
  MPW
    special considerations 1-13
mxArray 1-6, 3-7
  accessing data 2-3
  lr 1-6
Index

j c 1-6
nz max 1-6
parameters 1-6
pi 1-6
pr 1-6
structure 1-6
mxCalloc 2-11, 3-7
mxCopyPtrToComplex 3-20
mxCopyPtrToInt64 3-28
mxCopyPtrToReal 3-17, 3-28
mxCreateCellArray 2-20
mxCreateFull 3-7
mxCreateNumericArray 2-25, 2-27
mxCreateSparse 3-7
mxCreateString 3-7
mxCreateStructArray 2-15
mxGetCell 2-20
mxGetField 2-15
mxGetPi 2-23, 2-25, 2-27, 3-14, 3-20
mxGetPr 2-23, 2-25, 2-27, 3-14, 3-20
mxGetString 2-15
mxSetCell 2-20
mxSetField 2-15

N
nlhs 2-2, 2-5, 3-6
nrhs 2-2, 2-5, 3-3, 3-6
nz max 1-6, 3-26

O
options file
Macintosh 1-10
modifying 6-8
preconfigured 1-14
selecting Macintosh default 1-11

P
pi 1-6
plhs 2-2, 2-5, 3-3, 3-6
pointer 3-5
Fortran language MEX-file 3-14
pr 1-6
prhs 2-2, 2-5, 3-3, 3-6
protocol
DCOM 5-13
PutFullMatrix 5-11

R
registry
Windows 5-8
routine
computational 2-2
gateway 2-2

S
save 4-5
scope of variables
M- and MEX-file differences 2-32, 3-32
service name 5-15
sparse array 2-28
structures 2-15
system configuration 1-9
T
troubleshooting
  configuration 1-16
  MEX-file creation 6-2

V
variable scope 2-32

W
Watcom compiler
  debugging 2-34
Windows
  ActiveX 5-8
  automation 5-8
  registry 5-8
Windows clipboard format
  metafilepict 5-16
  text 5-15
  XL table 5-16