4. Locate the `.LIB` files for the compiler version you are using under `matlabroot\extern\lib\win32\microsoft`. For example, for version 6.0, these files are in the `msvc60` subdirectory.

5. From this directory, add `libmx.lib`, `libmex.lib`, and `libmat.lib` to the library modules in the `LINK` settings option.

6. Add the MATLAB `include` directory, `MATLAB\EXTERN\INCLUDE` to the `include` path in the **Settings C/C++ Preprocessor** option.

7. Add `MATLAB_MEX_FILE` to the **C/C++ Preprocessor** option by selecting **Settings** from the **Build** menu, selecting **C/C++**, and then typing `,MATLAB_MEX_FILE` after the last entry in the **Preprocessor definitions** field.

8. To debug the MEX-file using the IDE, put `MATLAB.EXE` in the **Settings Debug** option as the **Executable for debug session**.

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

Using the Add-In for Visual Studio

The MathWorks provides a MATLAB add-in for the Visual Studio® development system that lets you work easily within Microsoft Visual C/C++ (MSVC). The MATLAB add-in for Visual Studio greatly simplifies using M-files in the MSVC environment. The add-in automates the integration of M-files into Visual C++ projects. It is fully integrated with the MSVC environment.

The add-in for Visual Studio is automatically installed on your system when you run either `mbuild -setup` or `mex -setup` and select Microsoft Visual C/C++ version 5 or 6. However, there are several steps you must follow in order to use the add-in:

1 To build MEX-files with the add-in for Visual Studio, run the following command at the MATLAB command prompt.

```
mex -setup
```

Follow the menus and choose either Microsoft Visual C/C++ 5.0 or 6.0. This configures `mex` to use the selected Microsoft compiler and also installs the necessary add-in files in your Microsoft Visual C/C++ directories.

2 To configure the MATLAB add-in for Visual Studio to work with Microsoft Visual C/C++:

a Select **Tools** -> **Customize** from the MSVC menu.

b Click on the **Add-ins and Macro Files** tab.

c Check **MATLAB for Visual Studio** on the **Add-ins and Macro Files** list and click **Close**. The floating MATLAB add-in for Visual Studio toolbar appears. The checkmark directs MSVC to automatically load the add-in when you start MSVC again.

Configuring on Windows 98 and Windows ME Systems

Windows 98. To run the MATLAB add-in for Visual Studio on Windows 98 systems, add this line to your `config.sys` file.

```
shell=c:\command.com /e:32768 /p
```

Windows ME. To run the MATLAB add-in for Visual Studio on Windows ME systems, do the following:

1 Find `C:\windows\system\conagent.exe` in the Windows Explorer.

2 Right click on the `conagent.exe` icon.

3 Select **Properties** from the context menu. This brings up the **CONAGENT.EXE Properties** window.

4 Select the **Memory** tab in the **CONAGENT.EXE Properties** window.

5 Set the **Initial Environment** field to 4096.

6 Click **Apply**.

Ex.1049 / Page 347 of 748
Intel Corporation et al

7 Click **OK**.

For additional information on the MATLAB add-in for Visual Studio:

- See the `MATLABAddin.hlp` file in the `<matlab>\bin\win32` directory, or
- Click on the Help icon in the MATLAB add-in for Visual Studio toolbar.



Troubleshooting

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

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

Configuration Issues

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

Search Path Problem on Windows

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

MATLAB Pathnames Containing Spaces on Windows

If you have problems building MEX-files on Windows and there is a space in any of the directory names within the MATLAB path, you need to either reinstall MATLAB into a pathname that contains no spaces or rename the directory that contains the space. For example, if you install MATLAB under the `Program Files` directory, you may have difficulty building MEX-files with certain C compilers. Also, if you install MATLAB in a directory such as `MATLAB V5.2`, you may have difficulty.

DLLs Not on Path on Windows

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

Internal Error When Using mex -setup (PC).

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

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

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

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

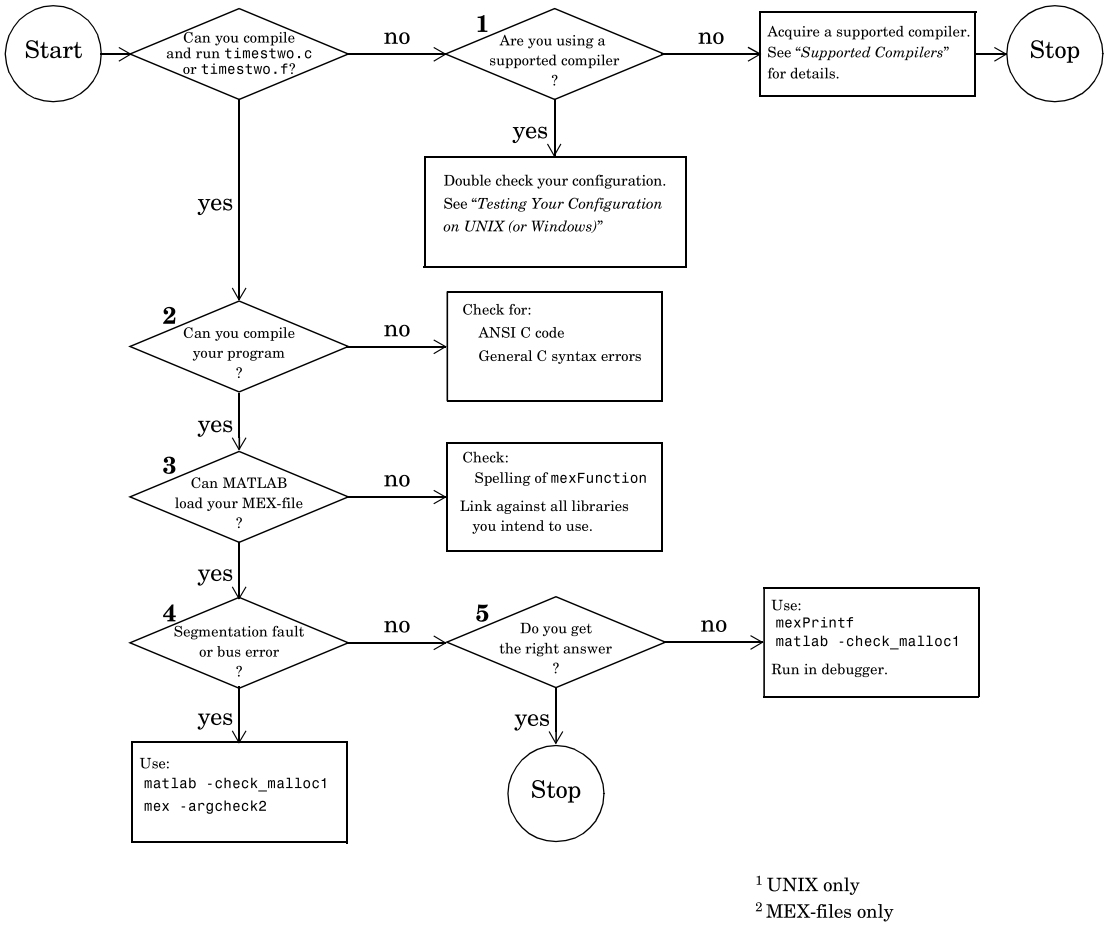
General Configuration Problem

Make sure you followed the configuration steps for your platform described in this chapter. Also, refer to “Custom Building MEX-Files” on page 3-19 for additional information.

Ex.1049 / Page 350 of 748
Intel Corporation et al

Understanding MEX-File Problems

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



[1] UNIX only
[2] MEX-files only

Figure 3-1: Troubleshooting MEX-File Creation Problems

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

Problem 1 - Compiling a MathWorks Program Fails

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

A reliable way of knowing if you have this type of configuration problem is if the header files supplied by The MathWorks generate a string of syntax errors when you try to compile your code. See “Building MEX-Files” on page 3-10 for information on selecting the appropriate options file or, if necessary, obtain an ANSI C compiler.

Problem 2 - Compiling Your Own Program Fails

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

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

Problem 3 - MEX-File Load Errors

If you receive an error of the form

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

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

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

On some platforms, if you fail to link against required libraries, you may get an error when MATLAB loads your MEX-file rather than when you compile your

Ex.1049 / Page 352 of 748
Intel Corporation et al

MEX-file. In such cases, you see a system error message referring to *unresolved symbols* or *unresolved references*. Be sure to link against the library that defines the function in question.

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

Problem 4 - Segmentation Fault or Bus Error

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

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

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

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

  Although your MEX-file will not run as efficiently as it can, this switch detects such errors as passing `null` pointers to API functions.
- **Run MATLAB with the -check_malloc option (UNIX only).** The MATLAB startup flag, `-check_malloc`, indicates that MATLAB should maintain additional memory checking information. When memory is freed, MATLAB checks to make sure that memory just before and just after this memory remains unwritten and that the memory has not been previously freed.

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

Ex.1049 / Page 353 of 748
Intel Corporation et al

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

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

Problem 5 - Program Generates Incorrect Results

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

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

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

Compiler and Platform-Specific Issues

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

MEX-Files Created in Watcom IDE

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

Memory Management Compatibility Issues

MATLAB now implicitly calls `mxDestroyArray`, the `mxArray` destructor, at the end of a MEX-file's execution on any `mxArrays` that are not returned in the left-hand side list (`plhs[]`). MATLAB issues a warning when it detects any misconstructed or improperly destructed `mxArrays`.

Ex.1049 / Page 354 of 748
Intel Corporation et al

We highly recommend that you fix code in your MEX-files that produces any of the warnings discussed in the following sections. For additional information, see “Memory Management” on page 4-37 in Creating C Language MEX-Files.

Note Currently, the following warnings are enabled by default for backwards compatibility reasons. In future releases of MATLAB, the warnings will be disabled by default. The programmer will be responsible for enabling these warnings during the MEX-file development cycle.

Improperly Destroying an mxArray

You cannot use `mxFree` to destroy an `mxArray`.

Warning

```
Warning:  You are attempting to call mxFree on a <class-id> array.
The destructor for mxArrays is mxDestroyArray; please call this
instead. MATLAB will attempt to fix the problem and continue, but
this will result in memory faults in future releases.
```

Example That Causes Warning

In the following example, `mxFree` does not destroy the array object. This operation frees the structure header associated with the array, but MATLAB will still operate as if the array object needs to be destroyed. Thus MATLAB will try to destroy the array object, and in the process, attempt to free its structure header again.

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

Solution

Call `mxDestroyArray` instead.

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


Ex.1049 / Page 355 of 748
Intel Corporation et al

Incorrectly Constructing a Cell or Structure mxArray

You cannot call `mxSetCell` or `mxSetField` variants with `prhs[]` as the member array.

Warning

```
Warning: You are attempting to use an array from another scope
(most likely an input argument) as a member of a cell array or
structure. You need to make a copy of the array first. MATLAB will
attempt to fix the problem and continue, but this will result in
memory faults in future releases.
```

Example That Causes Warning

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

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

    mxArray *temp = mxCreateCellMatrix(1,1);
      ...
    mxSetCell(temp, 0, prhs[0]);  /* INCORRECT */
```

Solution

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

```
mxSetCell(temp, 0, mxDuplicateArray(prhs[0]));  /* CORRECT */
```

Creating a Temporary mxArray with Improper Data

You cannot call `mxDestroyArray` on an `mxArray` whose data was not allocated by an API routine.

Ex.1049 / Page 356 of 748
Intel Corporation et al

Warning

```
Warning: You have attempted to point the data of an array to a
block of memory not allocated through the MATLAB API. MATLAB will
attempt to fix the problem and continue, but this will result in
memory faults in future releases.
```

Example That Causes Warning

If you call `mxSetPr`, `mxSetPi`, `mxSetData`, or `mxSetImagData`, specifying memory that was not allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc` as the intended data block (second argument), then when the MEX-file returns, MATLAB will attempt to free the pointer to real data and the pointer to imaginary data (if any). Thus MATLAB will attempt to free memory, in this example, from the program stack. This will cause the above warning when MATLAB attempts to reconcile its consistency checking information.

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

Solution

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

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

Potential Memory Leaks

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

Ex.1049 / Page 357 of 748
Intel Corporation et al

For example,

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

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

You can avoid that memory leak by changing the code

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

or alternatively

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

Note that the first solution is more efficient.

Similar memory leaks can also occur when using `mxSetPi`, `mxSetData`, `mxSetImagData`, `mxSetIr`, or `mxSetJc`. You can address this issue as shown above to avoid such memory leaks.

MEX-Files Should Destroy Their Own Temporary Arrays

In general, we recommend that MEX-files destroy their own temporary arrays and clean up their own temporary memory. All `mxArrays` except those returned in the left-hand side list and those returned by `mexGetVariablePtr` may be safely destroyed. This approach is consistent with other MATLAB API applications (i.e., MAT-file applications, engine applications, and MATLAB Compiler generated applications, which do not have any automatic cleanup mechanism.)

Additional Information

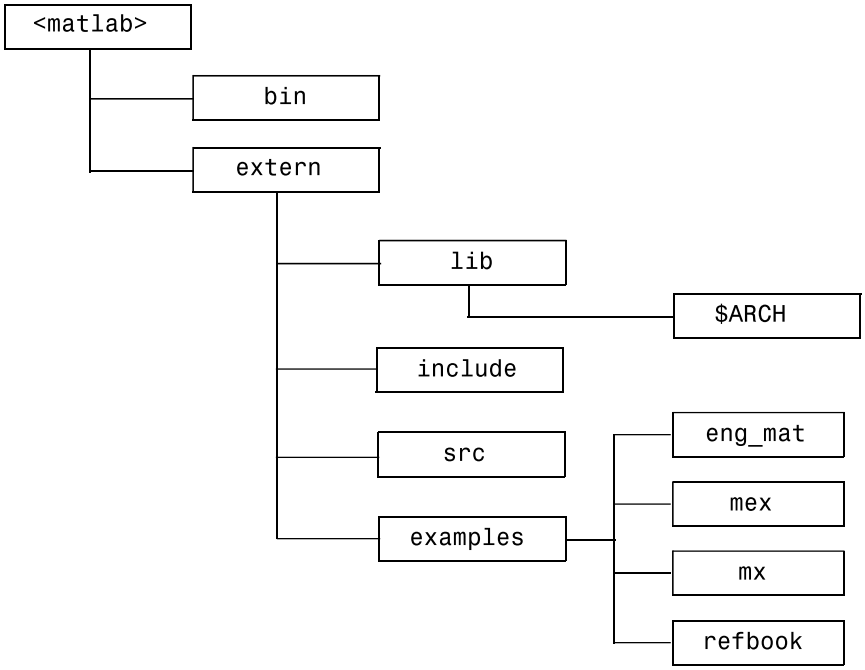
The following sections describe how to find additional information and assistance in building your applications. It covers the following topics:

- “Files and Directories - UNIX Systems”
- “Files and Directories - Windows Systems”
- “Examples”
- “Technical Support”

Files and Directories - UNIX Systems

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

The following figure illustrates the directories in which the MATLAB API files are located. In the illustration, `<matlab>` symbolizes the top-level directory where MATLAB is installed on your system.



<matlab>/bin

The `<matlab>/bin` directory contains two files that are relevant for the MATLAB API.

`mex`	UNIX shell script that creates MEX-files from C or Fortran MEX-file source code.
`matlab`	UNIX shell script that initializes your environment and then invokes the MATLAB interpreter.

This directory also contains the preconfigured options files that the `mex` script uses with particular compilers. This table lists the options files.

Table 3-4: Preconfigured Options Files

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

<matlab>/extern/lib/$ARCH

The `<matlab>/extern/lib/`*`$ARCH`* directory contains libraries, where *`$ARCH`* specifies a particular UNIX platform. On some UNIX platforms, this directory contains two versions of this library. Library filenames ending with `.a` are static libraries and filenames ending with `.so` or `.sl` are shared libraries.

Ex.1049 / Page 360 of 748
Intel Corporation et al

<matlab>/extern/include

The `<matlab>/extern/include` directory contains the header files for developing C and C++ applications that interface with MATLAB.

The relevant header files for the MATLAB API are:

`engine.h`	Header file for MATLAB engine programs. Contains function prototypes for engine routines.
`mat.h`	Header file for programs accessing MAT-files. Contains function prototypes for `mat` routines.
`matrix.h`	Header file containing a definition of the `mxArray` structure and function prototypes for matrix access routines.
`mex.h`	Header file for building MEX-files. Contains function prototypes for `mex` routines.

<matlab>/extern/src

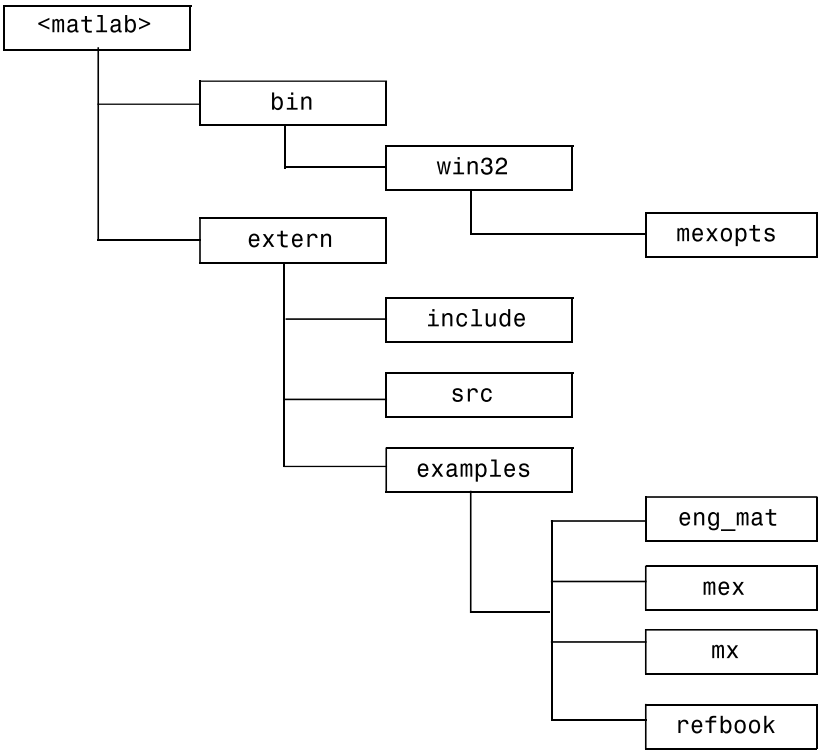
The `<matlab>/extern/src` directory contains those C source files that are necessary to support certain MEX-file features such as argument checking and versioning.

Files and Directories - Windows Systems

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

The following figure illustrates the directories in which the MATLAB API files are located. In the illustration, <matlab> symbolizes the top-level directory where MATLAB is installed on your system.

Ex.1049 / Page 361 of 748
Intel Corporation et al



<matlab>\bin\win32

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

<matlab>\bin\win32\mexopts

The `<matlab>\bin\win32\mexopts` directory contains the preconfigured options files that the `mex` script uses with particular compilers. See Table 3-2, Options Files, on page 3-16 for a complete list of the options files.

Ex.1049 / Page 362 of 748
Intel Corporation et al

<matlab>\extern\include

The <matlab>\extern\include directory contains the header files for developing C and C++ applications that interface with MATLAB.

The relevant header files for the MATLAB API (MEX-files, engine, and MAT-files) are:

engine.h	Header file for MATLAB engine programs. Contains function prototypes for engine routines.
mat.h	Header file for programs accessing MAT-files. Contains function prototypes for mat routines.
matrix.h	Header file containing a definition of the mxArray structure and function prototypes for matrix access routines.
mex.h	Header file for building MEX-files. Contains function prototypes for mex routines.
_*.def	Files used by Borland compiler.
*.def	Files used by MSVC and Microsoft Fortran compilers.
mexversion.rc	Resource file for inserting versioning information into MEX-files.

<matlab>\extern\src

The <matlab>\extern\src directory contains files that are used for debugging MEX-files.

Examples

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

Examples From the Text

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

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

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

MEX Reference Examples

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

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

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

MX Examples

The `mx` subdirectory of `extern/examples` contains examples for using the array access functions. Although you can use these functions in stand-alone programs, most of these are MEX-file examples. The exception is `mxSetAllocFcns.c`, since this function is available only to stand-alone programs.

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

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

Engine and MAT Examples

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

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

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


Ex.1049 / Page 364 of 748
Intel Corporation et al

Technical Support

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

- FAQ (Frequently Asked Questions)

This is a list of the most frequently asked questions received by our Technical Support staff, along with answers from the solutions database.

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

- Solution Search Engine

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

```
http://www.mathworks.com/support/solutions/index.shtml
```

- Technical Notes

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

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

Ex.1049 / Page 366 of 748
Intel Corporation et al

4

Creating C Language MEX-Files

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

C MEX-Files (p. 4-2)	MEX-file components and required arguments
Examples of C MEX-Files (p. 4-6)	Sample MEX-files that show how to handle all data types
Advanced Topics (p. 4-36)	Help files, linking multiple files, workspace, managing memory, using LAPACK and BLAS functions
Debugging C Language MEX-Files (p. 4-48)	Debugging MEX-file srouce code from within MATLAB

C MEX-Files

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

The Components of a C MEX-File

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

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

In the gateway routine, you can access the data in the `mxArray` structure and then manipulate this data in your C computational subroutine. For example, the expression `mxGetPr(prhs[0])` returns a pointer of type `double *` to the real data in the `mxArray` pointed to by `prhs[0]`. You can then use this pointer like any other pointer of type `double *` in C. After calling your C computational routine from the gateway, you can set a pointer of type `mxArray` to the data it returns. MATLAB is then able to recognize the output from your computational routine as the output from the MEX-file.

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

Ex.1049 / Page 368 of 748
Intel Corporation et al

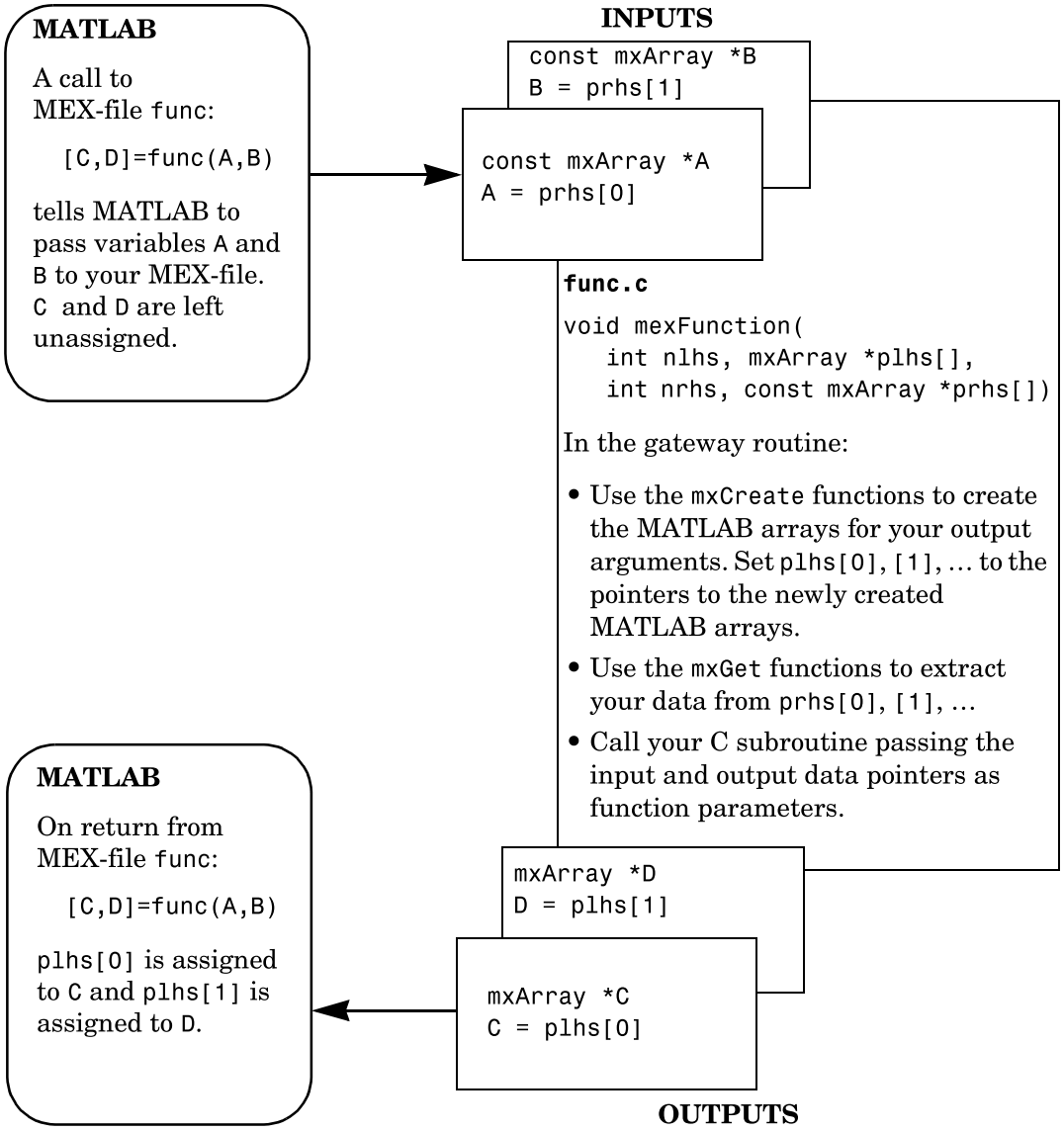


Figure 4-1: C MEX Cycle

Ex.1049 / Page 369 of 748
Intel Corporation et al

Required Arguments to a MEX-File

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

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

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

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

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

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

For example, if you invoke a MEX-file from the MATLAB workspace with the command

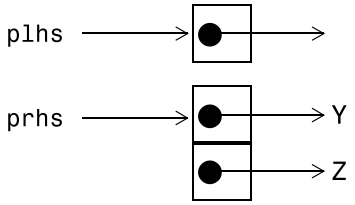
```
x = fun(y,z);
```


Ex.1049 / Page 370 of 748
Intel Corporation et al

the MATLAB interpreter calls `mexFunction` with the arguments

```
nlhs = 1

nrhs = 2
```



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

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

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

Ex.1049 / Page 371 of 748
Intel Corporation et al

Examples of C MEX-Files

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

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

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

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

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

Ex.1049 / Page 372 of 748
Intel Corporation et al

A First Example — Passing a Scalar

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

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

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

```
/*
 * ============================================================
 * timestwo.c - example found in API guide
 *
 * Computational function that takes a scalar and doubles it.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * ============================================================
 */

/* $Revision: 1.8 $ */

#include "mex.h"

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


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

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

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

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

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

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

In C, function argument checking is done at compile time. In MATLAB, you can pass any number or type of arguments to your M-function, which is responsible for argument checking. This is also true for MEX-files. Your program must safely handle any number of input or output arguments of any supported type.

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

```
mex timestwo.c
```

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

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

You can create and compile MEX-files in MATLAB or at your operating system's prompt. MATLAB uses `mex.m`, an M-file version of the `mex` script, and your operating system uses `mex.bat` on Windows and `mex.sh` on UNIX. In either case, typing

```
mex filename
```

at the prompt produces a compiled version of your MEX-file.

In the above example, scalars are viewed as 1-by-1 matrices. Alternatively, you can use a special API function called `mxGetScalar` that returns the values of scalars instead of pointers to copies of scalar variables. This is the alternative code (error checking has been omitted for brevity).

```
/*
 * ==============================================================
 * timestwoalt.c - example found in API guide
 *
 * Use mxGetScalar to return the values of scalars instead of
 * pointers to copies of scalar variables.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * ==============================================================
 */

/* $Revision: 1.5 $ */

#include "mex.h"

void timestwo_alt(double *y, double x)
{
  *y = 2.0*x;
}
```

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

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

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

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

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

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

Passing Strings

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

```
/*
 * ================================================================
 * revord.c
 * Example for illustrating how to copy the string data from
 * MATLAB to a C-style string and back again.
 *
 * Takes a string and returns a string in reverse order.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * ================================================================
 */
```

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

#include "mex.h"

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

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

In this example, the API function `mxCalloc` replaces `calloc`, the standard C function for dynamic memory allocation. `mxCalloc` allocates dynamic memory using the MATLAB memory manager and initializes it to zero. You must use `mxCalloc` in any situation where C would require the use of `calloc`. The same is true for `mxMalloc` and `mxRealloc`; use `mxMalloc` in any situation where C would require the use of `malloc` and use `mxRealloc` where C would require `realloc`.

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

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

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

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

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

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

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

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

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

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

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

The gateway routine allocates memory for the input and output strings. Since these are C strings, they need to be one greater than the number of elements in the MATLAB string. Next the MATLAB string is copied to the input string. Both the input and output strings are passed to the computational subroutine (`revord`), which loads the output in reverse order. Note that the output buffer is a valid null-terminated C string because `mxCalloc` initializes the memory to 0. The API function `mxCreateString` then creates a MATLAB string from the

C string, `output_buf`. Finally, `plhs[0]`, the left-hand side return argument to MATLAB, is set to the MATLAB array you just created.

By isolating variables of type `mxArray` from the computational subroutine, you can avoid having to make significant changes to your original C code.

In this example, typing

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

produces

```
The string to convert is 'hello world'.
y =
dlrow olleh
```

Passing Two or More Inputs or Outputs

The `plhs[]` and `prhs[]` parameters are vectors that contain pointers to each left-hand side (output) variable and each right-hand side (input) variable, respectively. Accordingly, `plhs[0]` contains a pointer to the first left-hand side argument, `plhs[1]` contains a pointer to the second left-hand side argument, and so on. Likewise, `prhs[0]` contains a pointer to the first right-hand side argument, `prhs[1]` points to the second, and so on.

This example, `xtimesy`, multiplies an input scalar by an input scalar or matrix and outputs a matrix. For example, using `xtimesy` with two scalars gives

```
x = 7;
y = 7;
z = xtimesy(x,y)

z =
     49
```

Using `xtimesy` with a scalar and a matrix gives

```
x = 9;
y = ones(3);
z = xtimesy(x,y)
```


Ex.1049 / Page 379 of 748
Intel Corporation et al

```
z =
     9     9     9
     9     9     9
     9     9     9
```

This is the corresponding MEX-file C code.

```
/*
 * ================================================================
 * xtimesy.c - example found in API guide
 *
 * Multiplies an input scalar times an input matrix and outputs a
 * matrix.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * ================================================================
 */

/* $Revision: 1.10 $ */

#include "mex.h"

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

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

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

  /*  Check for proper number of arguments. */
  /* NOTE: You do not need an else statement when using
     mexErrMsgTxt within an if statement. It will never
     get to the else statement if mexErrMsgTxt is executed.
     (mexErrMsgTxt breaks you out of the MEX-file.)
  */
  if (nrhs != 2)
    mexErrMsgTxt("Two inputs required.");
  if (nlhs != 1)
    mexErrMsgTxt("One output required.");

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

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

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

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

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

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

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

As this example shows, creating MEX-file gateways that handle multiple inputs and outputs is straightforward. All you need to do is keep track of which indices of the vectors `prhs` and `plhs` correspond to the input and output arguments of your function. In the example above, the input variable `x` corresponds to `prhs[0]` and the input variable `y` to `prhs[1]`.

Note that `mxGetScalar` returns the value of `x` rather than a pointer to `x`. This is just an alternative way of handling scalars. You could treat `x` as a 1-by-1 matrix and use `mxGetPr` to return a pointer to `x`.

Passing Structures and Cell Arrays

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

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

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

```
/*
 * ===============================================================
 * phonebook.c
 * Example for illustrating how to manipulate structure and cell
 * array
 *
 * Takes a (MxN) structure matrix and returns a new structure
 * (1x1) containing corresponding fields:for string input, it
 * will be (MxN) cell array; and for numeric (noncomplex, scalar)
 * input, it will be (MxN) vector of numbers with the same
```

```
 * classID as input, such as int, double etc..
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * ============================================================
 */

/* $Revision: 1.6 $ */

#include "mex.h"
#include "string.h"

#define MAXCHARS 80   /* max length of string contained in each
                         field */

/* The gateway routine.  */
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  const char **fnames;       /* pointers to field names */
  const int  *dims;
  mxArray    *tmp, *fout;
  char       *pdata;
  int        ifield, jstruct, *classIDflags;
  int        NStructElems, nfields, ndim;

  /* Check proper input and output */
  if (nrhs != 1)
    mexErrMsgTxt("One input required.");
  else if (nlhs > 1)
    mexErrMsgTxt("Too many output arguments.");
  else if (!mxIsStruct(prhs[0]))
    mexErrMsgTxt("Input must be a structure.");

  /* Get input arguments */
  nfields = mxGetNumberOfFields(prhs[0]);
  NStructElems = mxGetNumberOfElements(prhs[0]);

  /* Allocate memory  for storing classIDflags */
  classIDflags = mxCalloc(nfields, sizeof(int));
```

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

/* Allocate memory  for storing pointers */
fnames = mxCalloc(nfields, sizeof(*fnames));
```

```
  /* Get field name pointers */
  for (ifield = 0; ifield < nfields; ifield++) {
    fnames[ifield] = mxGetFieldNameByNumber(prhs[0],ifield);
  }

  /* Create a 1x1 struct matrix for output */
  plhs[0] = mxCreateStructMatrix(1, 1, nfields, fnames);
  mxFree(fnames);
  ndim = mxGetNumberOfDimensions(prhs[0]);
  dims = mxGetDimensions(prhs[0]);
  for (ifield = 0; ifield < nfields; ifield++) {
    /* Create cell/numeric array */
    if (classIDflags[ifield] == mxCHAR_CLASS) {
      fout = mxCreateCellArray(ndim, dims);
    } else {
      fout = mxCreateNumericArray(ndim, dims,
          classIDflags[ifield], mxREAL);
      pdata = mxGetData(fout);
    }

    /* Copy data from input structure array */
    for (jstruct = 0; jstruct < NStructElems; jstruct++) {
      tmp = mxGetFieldByNumber(prhs[0],jstruct,ifield);
      if (mxIsChar(tmp)) {
        mxSetCell(fout, jstruct, mxDuplicateArray(tmp));
      } else {
        size_t     sizebuf;
        sizebuf = mxGetElementSize(tmp);
        memcpy(pdata, mxGetData(tmp), sizebuf);
        pdata += sizebuf;
      }
    }

    /* Set each field in output structure */
    mxSetFieldByNumber(plhs[0], 0, ifield, fout);
  }
  mxFree(classIDflags);
  return;
}
```

To see how this program works, enter this structure.

```
friends(1).name = 'Jordan Robert';
friends(1).phone = 3386;
friends(2).name = 'Mary Smith';
friends(2).phone = 3912;
friends(3).name = 'Stacy Flora';
friends(3).phone = 3238;
friends(4).name = 'Harry Alpert';
friends(4).phone = 3077;
```

The results of this input are

```
phonebook(friends)

ans =
     name: {1x4 cell  }
    phone: [3386 3912 3238 3077]
```

Handling Complex Data

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

This example takes two complex row vectors and convolves them.

```
/*
 * ==============================================================
 * convec.c
 * Example for illustrating how to pass complex data
 * from MATLAB to C and back again
 *
 * Convolves two complex input vectors.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * ==============================================================
 */

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

```
#include "mex.h"

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

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

Below is the gateway routine that calls this complex convolution.

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

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

  /* Check that both inputs are row vectors. */
  if (mxGetM(prhs[0]) != 1 || mxGetM(prhs[1]) != 1)
    mexErrMsgTxt("Both inputs must be row vectors.");
  rows = 1;
```


Ex.1049 / Page 387 of 748
Intel Corporation et al

```
    /* Check that both inputs are complex. */
    if (!mxIsComplex(prhs[0]) || !mxIsComplex(prhs[1]))
      mexErrMsgTxt("Inputs must be complex.\n");

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

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

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

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

    return;
  }
```

Entering these numbers at the MATLAB prompt

```
x = [3.000 - 1.000i, 4.000 + 2.000i, 7.000 - 3.000i];
y = [8.000 - 6.000i, 12.000 + 16.000i, 40.000 - 42.000i];
```

and invoking the new MEX-file

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

results in

```
z =
   1.0e+02 *
```

```
Columns 1 through 4

0.1800 - 0.2600i 0.9600 + 0.2800i 1.3200 - 1.4400i 3.7600 - 0.1200i

Column 5

1.5400 - 4.1400i
```

which agrees with the results that the built-in MATLAB function `conv.m` produces.

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

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

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

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

```
/*
 * ============================================================
 * doubleelement.c - Example found in API Guide
 *
 * Constructs a 2-by-2 matrix with unsigned 16-bit integers,
 * doubles each element, and returns the matrix.
 *
```

```
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * ================================================================
 */

/* $Revision: 1.9 $ */

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

#define NDIMS 2
#define TOTAL_ELEMENTS 4

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

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


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

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

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

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

At the MATLAB prompt, entering

```
doubleelement
```

produces

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

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

Manipulating Multidimensional Numerical Arrays

You can manipulate multidimensional numerical arrays by using `mxGetData` and `mxGetImagData` to return pointers to the real and imaginary parts of the data stored in the original multidimensional array. This example takes an N-dimensional array of doubles and returns the indices for the nonzero elements in the array.

```
/*
 * ================================================================
 * findnz.c
 * Example for illustrating how to handle N-dimensional arrays in
 * a MEX-file. NOTE: MATLAB uses 1-based indexing, C uses 0-based
 * indexing.
 *
 * Takes an N-dimensional array of doubles and returns the indices
 * for the non-zero elements in the array. findnz works
 * differently than the FIND command in MATLAB in that it returns
```

```
 * all the indices in one output variable, where the column
 * element contains the index for that dimension.
 *
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 by The MathWorks, Inc.
 * ==============================================================
 */

/* $Revision: 1.5 $ */

#include "mex.h"

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

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

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

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

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

/* Get the number of elements in the input argument. */
elements = mxGetNumberOfElements(prhs[0]);

/* Get the data. */
pr = (double *)mxGetPr(prhs[0]);
pi = (double *)mxGetPi(prhs[0]);
cmplx = ((pi == NULL) ? 0 : 1);

/* Count the number of non-zero elements to be able to allocate
   the correct size for output variable. */
for (j = 0; j < elements; j++) {
  if (IsNonZero(pr[j]) || (cmplx && IsNonZero(pi[j]))) {
    nnz++;
  }
}

/* Get the number of dimensions in the input argument.
   Allocate the space for the return argument */
number_of_dims = mxGetNumberOfDimensions(prhs[0]);
plhs[0] = mxCreateDoubleMatrix(nnz, number_of_dims, mxREAL);
pind = mxGetPr(plhs[0]);

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

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

Entering a sample matrix at the MATLAB prompt gives

```
matrix = [ 3 0 9 0; 0 8 2 4; 0 9 2 4; 3 0 9 3; 9 9 2 0]
matrix =
     3     0     9     0
     0     8     2     4
     0     9     2     4
     3     0     9     3
     9     9     2     0
```

This example determines the position of all nonzero elements in the matrix. Running the MEX-file on this matrix produces

```
nz = findnz(matrix)
nz =
     1     1
     4     1
     5     1
     2     2
     3     2
     5     2
     1     3
```

```
     2     3
     3     3
     4     3
     5     3
     2     4
     3     4
     4     4
```

Handling Sparse Matrices

The MATLAB API provides a set of functions that allow you to create and manipulate sparse arrays from within your MEX-files. These API routines access and manipulate `ir` and `jc`, two of the parameters associated with sparse arrays. For more information on how MATLAB stores sparse arrays, refer to the section, "The MATLAB Array" on page 3-5.

This example illustrates how to populate a sparse matrix.

```
/*
 * ==============================================================
 * fulltosparse.c
 * This example demonstrates how to populate a sparse
 * matrix.  For the purpose of this example, you must pass in a
 * non-sparse 2-dimensional argument of type double.
 *
 * Comment: You might want to modify this MEX-file so that you can
 * use it to read large sparse data sets into MATLAB.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * ==============================================================
 */

/* $Revision: 1.5 $ */

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

/* If you are using a compiler that equates NaN to be zero, you
 * must compile this example using the flag  -DNAN_EQUALS_ZERO.
 * For example:
```

```
 *
 *     mex -DNAN_EQUALS_ZERO fulltosparse.c
 *
 * This will correctly define the IsNonZero macro for your C
 * compiler.
 */

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

void mexFunction(
        int nlhs,       mxArray *plhs[],
        int nrhs, const mxArray *prhs[]
        )
{
  /* Declare variables. */
  int j,k,m,n,nzmax,*irs,*jcs,cmplx,isfull;
  double *pr,*pi,*si,*sr;
  double percent_sparse;

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

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

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

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

/* Allocate space for sparse matrix.
 * NOTE:  Assume at most 20% of the data is sparse.  Use ceil
 * to cause it to round up.
 */

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

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

/* Copy nonzeros. */
k = 0;
isfull = 0;
for (j = 0; (j < n); j++) {
  int i;
  jcs[j] = k;
  for (i = 0; (i < m); i++) {
    if (IsNonZero(pr[i]) || (cmplx && IsNonZero(pi[i]))) {

      /* Check to see if non-zero element will fit in
       * allocated output array.  If not, increase
       * percent_sparse by 10%, recalculate nzmax, and augment
       * the sparse array.
       */
      if (k >= nzmax) {
        int oldnzmax = nzmax;
        percent_sparse += 0.1;
        nzmax = (int)ceil((double)m*(double)n*percent_sparse);
```

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

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

            sr  = mxGetPr(plhs[0]);
            si  = mxGetPi(plhs[0]);
            irs = mxGetIr(plhs[0]);
          }
          sr[k] = pr[i];
          if (cmplx) {
            si[k] = pi[i];
          }
          irs[k] = i;
          k++;
        }
      }
      pr += m;
      pi += m;
    }
    jcs[n] = k;
  }
```

At the MATLAB prompt, entering

```
full = eye(5)
full =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

creates a full, 5-by-5 identity matrix. Using `fulltosparse` on the full matrix produces the corresponding sparse matrix.

Ex.1049 / Page 398 of 748
Intel Corporation et al

```
spar = fulltosparse(full)
spar =
   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
```

Calling Functions from C MEX-Files

It is possible to call MATLAB functions, operators, M-files, and other MEX-files from within your C source code by using the API function `mexCallMATLAB`. This example creates an `mxArray`, passes various pointers to a subfunction to acquire data, and calls `mexCallMATLAB` to calculate the sine function and plot the results.

```
/*
 * ============================================================
 * sincall.c
 *
 * Example for illustrating how to use mexCallMATLAB
 *
 * Creates an mxArray and passes its associated pointers (in
 * this demo, only pointer to its real part, pointer to number of
 * rows, pointer to number of columns) to subfunction fill() to
 * get data filled up, then calls mexCallMATLAB to calculate sin
 * function and plot the result.
 *
 * This is a MEX-file for MATLAB.
 * Copyright (c) 1984-2000 The MathWorks, Inc.
 * ============================================================
 */

/* $Revision: 1.4 $ */

#include "mex.h"
#define MAX 1000
```

```
/* Subroutine for filling up data */
void fill(double *pr, int *pm, int *pn, int max)
{
  int i;

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

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

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

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

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

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

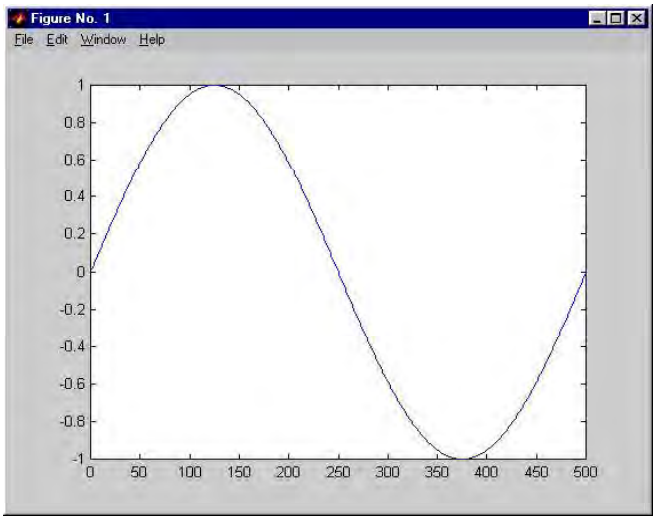
  return;
}
```


Ex.1049 / Page 400 of 748
Intel Corporation et al

Running this example

```
sincall
```

displays the results



Note It is possible to generate an object of type mxUNKNOWN_CLASS using `mexCallMATLAB`. See the example below.

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

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

MATLAB displays the following warning message.

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

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

Advanced Topics

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

Help Files

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

Linking Multiple Files

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

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

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

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

Note On UNIX, you must use the `-cxx` switch to the `mex` script if you are linking C++ objects.

Ex.1049 / Page 402 of 748
Intel Corporation et al

Workspace for MEX-File Functions

Unlike M-file functions, MEX-file functions do not have their own variable workspace. MEX-file functions operate in the caller's workspace.

`mexEvalString` evaluates the string in the caller's workspace. In addition, you can use the `mexGetVariable` and `mexPutVariable` routines to get and put variables into the caller's workspace.

Memory Management

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

Automatic Cleanup of Temporary Arrays

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

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

- A call to `mexErrMsgTxt` occurs.
- A call to `mexCallMATLAB` occurs and the function being called creates an error. (A MEX-file can trap such errors by using `mexSetTrapFlag`, but not all MEX-files would necessarily need to trap errors.)
- The user interrupts the MEX-file's execution using **Ctrl-C**.
- The MEX-file runs out of memory. When this happens, the MATLAB out-of-memory handler will immediately terminate the MEX-file.

A careful MEX-file programmer can ensure safe cleanup of all temporary arrays and memory before returning in the first two cases, but not in the last

two cases. In the last two cases, the automatic cleanup mechanism is necessary to prevent memory leaks.

Persistent Arrays

You can exempt an array, or a piece of memory, from the MATLAB automatic cleanup by calling `mexMakeArrayPersistent` or `mexMakeMemoryPersistent`. However, if a MEX-file creates such persistent objects, there is a danger that a memory leak could occur if the MEX-file is cleared before the persistent object is properly destroyed. In order to prevent this from happening, a MEX-file that creates persistent objects should register a function, using `mexAtExit`, which will dispose of the objects. (You can use a `mexAtExit` function to dispose of other resources as well; for example, you can use `mexAtExit` to close an open file.)

For example, here is a simple MEX-file that creates a persistent array and properly disposes of it.

```
#include "mex.h"

static int initialized = 0;
static mxArray *persistent_array_ptr = NULL;

void cleanup(void) {
    mexPrintf("MEX-file is terminating, destroying array\n");
    mxDestroyArray(persistent_array_ptr);
}

void mexFunction(int nlhs,
    mxArray *plhs[],
    int nrhs,
    const mxArray *prhs[])
{
  if (!initialized) {
    mexPrintf("MEX-file initializing, creating array\n");

    /* Create persistent array and register its cleanup. */
    persistent_array_ptr = mxCreateDoubleMatrix(1, 1, mxREAL);
    mexMakeArrayPersistent(persistent_array_ptr);
    mexAtExit(cleanup);
    initialized = 1;
```


Ex.1049 / Page 404 of 748
Intel Corporation et al

```
    /* Set the data of the array to some interesting value. */
    *mxGetPr(persistent_array_ptr) = 1.0;
  } else {
    mexPrintf("MEX-file executing; value of first array
        element is %g\n", *mxGetPr(persistent_array_ptr));
  }
}
```

Hybrid Arrays

Functions such as `mxSetPr`, `mxSetData`, and `mxSetCell` allow the direct placement of memory pieces into an `mxArray`. `mxDestroyArray` will destroy these pieces along with the entire array. Because of this, it is possible to create an array that cannot be destroyed, i.e., an array on which it is not safe to call `mxDestroyArray`. Such an array is called a *hybrid* array, because it contains both destroyable and nondestroyable components.

For example, it is not legal to call `mxFree` (or the ANSI `free()` function, for that matter) on automatic variables. Therefore, in the following code fragment, `pArray` is a hybrid array.

```
mxArray *pArray = mxCreateDoubleMatrix(0, 0, mxREAL);
double data[10];

mxSetPr(pArray, data);
mxSetM(pArray, 1);
mxSetN(pArray, 10);
```

Another example of a hybrid array is a cell array or structure, one of whose children is a read-only array (an array with the `const` qualifier, such as one of the inputs to the MEX-file). The array cannot be destroyed because the input to the MEX-file would also be destroyed.

Because hybrid arrays cannot be destroyed, they cannot be cleaned up by the automatic mechanism outlined in “Automatic Cleanup of Temporary Arrays” on page 4-37. As described in that section, the automatic cleanup mechanism is the only way to destroy temporary arrays in case of a user interrupt. Therefore, *temporary hybrid arrays are illegal* and may cause your MEX-file to crash.

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

Ex.1049 / Page 405 of 748
Intel Corporation et al

Using LAPACK and BLAS Functions

LAPACK is a large, multiauthor Fortran subroutine library that MATLAB uses for numerical linear algebra. BLAS, which stands for Basic Linear Algebra Subroutines, is used by MATLAB to speed up matrix multiplication and the LAPACK routines themselves. The functions provided by LAPACK and BLAS can also be called directly from within your C MEX-files.

This section explains how to write and build MEX-files that call LAPACK and BLAS functions. It provides information on

- "Specifying the Function Name"
- "Calling LAPACK and BLAS Functions from C"
- "Handling Complex Numbers" on page 4-41
- "Preserving Input Values from Modification" on page 4-43
- "Building the C MEX-File" on page 4-44
- "Example – Symmetric Indefinite Factorization Using LAPACK" on page 4-45
- "Calling LAPACK and BLAS Functions from Fortran" on page 4-45
- "Building the Fortran MEX-File" on page 4-46

Specifying the Function Name

When calling an LAPACK or BLAS function, some platforms require an underscore character following the function name in the call statement.

On the PC, IBM_RS, and HP platforms, use the function name alone, with no trailing underscore. For example, to call the LAPACK `dgemm` function, use

```
dgemm (arg1, arg2, ..., argn);
```

On the SGI, LINUX, Solaris, Alpha, and Macintosh platforms, add the underscore after the function name. For example, to call `dgemm` on any of these platforms, use

```
dgemm_ (arg1, arg2, ..., argn);
```

Calling LAPACK and BLAS Functions from C

Since the LAPACK and BLAS functions are written in Fortran, arguments passed to and from these functions must be passed by reference. The following

Ex.1049 / Page 406 of 748
Intel Corporation et al

example calls dgemm, passing all arguments by reference. An ampersand (&) precedes each argument unless that argument is already a reference.

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray
*prhs[])
{
  double *A, *B, *C, one = 1.0, zero = 0.0;
  int m,n,p;
  char *chn = "N";

  A = mxGetPr(prhs[0]);
  B = mxGetPr(prhs[1]);
  m = mxGetM(prhs[0]);
  p = mxGetN(prhs[0]);
  n = mxGetN(prhs[1]);

  if (p != mxGetM(prhs[1])) {
    mexErrMsgTxt("Inner dimensions of matrix multiply do not
        match");
  }

  plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
  C = mxGetPr(plhs[0]);

  /* Pass all arguments to Fortran by reference */
  dgemm (chn, chn, &m, &n, &p, &one, A, &m, B, &p, &zero, C, &m);
}
```

Handling Complex Numbers

MATLAB stores complex numbers differently than Fortran. MATLAB stores the real and imaginary parts of a complex number in separate, equal length vectors, pr and pi. Fortran stores the same number in one location with the real and imaginary parts interleaved.

As a result, complex variables exchanged between MATLAB and the Fortran functions in LAPACK and BLAS are incompatible. MATLAB provides conversion routines that change the storage format of complex numbers to address this incompatibility.

Input Arguments. For all complex variables passed as input arguments to a Fortran function, you need to convert the storage of the MATLAB variable to be compatible with the Fortran function. Use the `mat2fort` function for this. See the example that follows.

Output Arguments. For all complex variables passed as output arguments to a Fortran function, you need to do the following:

1. When allocating storage for the complex variable, allocate a real variable with twice as much space as you would for a MATLAB variable of the same size. You need to do this because the returned variable uses the Fortran format, which takes twice the space. See the allocation of `zout` in the example that follows.

2. Once the variable is returned to MATLAB, convert its storage so that it is compatible with MATLAB. Use the `fort2mat` function for this.

Example – Passing Complex Variables. The example below shows how to call an LAPACK function from MATLAB, passing complex `prhs[0]` as input and receiving complex `plhs[0]` as output. Temporary variables `zin` and `zout` are used to hold `prhs[0]` and `plhs[0]` in Fortran format.

```
#include "mex.h"
#include "fort.h"      /* defines mat2fort and fort2mat */

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray
*prhs[])
{
  int lda, n;
  double *zin, *zout;
  lda = mxGetM(prhs[0]);
  n = mxGetN(prhs[0]);

  /* Convert input to Fortran format */
  zin = mat2fort(prhs[0], lda, n);

 /* Allocate a real, complex, lda-by-n variable to store output */
  zout = mxCalloc(2*lda*n, sizeof(double));
```

```
  /* Call complex LAPACK function */
  zlapack_function(zin, &lda, &n, zout);

  /* Convert output to MATLAB format */
  plhs[0] = fort2mat(zout, lda, lda, n);

  /* Free intermediate Fortran format arrays */
  mxFree(zin);
  mxFree(zout);
}
```

Preserving Input Values from Modification

Many LAPACK and BLAS functions modify the values of arguments passed in to them. It is advisable to make a copy of arguments that can be modified prior to passing them to the function. For complex inputs, this point is moot since the `mat2fort` version of the input is a new piece of memory, but for real data this is not the case.

The following example calls an LAPACK function that modifies the first input argument. The code in this example makes a copy of `prhs[0]`, and then passes the copy to the LAPACK function to preserve the contents of `prhs[0]`.

```
/* lapack_function modifies A so make a copy of the input */
m = mxGetM(prhs[0]);
n = mxGetN(prhs[0]);
A = mxCalloc(m*n, sizeof(double));

/* Copy mxGetPr(prhs[0]) into A */
temp = mxGetPr(prhs[0]);
for (k = 0; k < m*n; k++) {
   A[k] = temp[k];
   }

/* lapack_function does not modify B so it is OK to use the input
directly */
B = mxGetPr(prhs[1]);
lapack_function(A, B);       /* modifies A but not B */

/* Free A when you are done with it */
mxFree(A);
```

Building the C MEX-File

The examples in this section show how to compile and link a C MEX file, `myCmexFile.c`, on the platforms supported by MATLAB. In each example, the term `<matlab>` stands for the MATLAB root directory.

Building on the PC. If you build your C MEX-file on a PC or IBM_RS platform, you need to explicitly specify a library file to link with.

On the PC, use this command if you are using the Lcc compiler that ships with MATLAB:

```
mex myCmexFile.c <matlab>/extern/lib/win32/lcc/libmwlapack.lib
```

Or, use this command if you are using Microsoft Visual C++ as your C compiler:

```
mex myCmexFile.c
<matlab>/extern/lib/win32/microsoft/msvc60/libmwlapack.lib
```

Building on IBM_RS. For IBM_RS, use the following syntax.

```
mex myCmexFile.c -L<matlab>/bin/ibm_rs -lmwlapack
```

Building on Other Platforms. On all other platforms, you can build your MEX-file as you would any other C MEX-file. For example,

```
mex myCmexFile.c
```

MEX-Files That Perform Complex Number Conversion. MATLAB supplies the files `fort.c` and `fort.h`, which provide routines for conversion between MATLAB and FORTRAN complex data structures. These files define the `mat2fort` and `fort2mat` routines mentioned previously under “Handling Complex Numbers” on page 4-41.

If your program uses these routines, then you need to:

1. Include the `fort.h` file in your program, using, `#include "fort.h"`. See the example above.

2. Build the `fort.c` file with your program. Specify the pathname, `<matlab>/extern/examples/refbook` for both `fort.c` and `fort.h` in the build command, (where `<matlab>` stands for the MATLAB root directory).

Ex.1049 / Page 410 of 748
Intel Corporation et al

On the PC, use either one of the following.

```
mex myCmexFile.c <matlab>/extern/examples/refbook/fort.c
-I<matlab>/extern/examples/refbook
<matlab>/extern/lib/win32/microsoft/msvc60/libmwlapack.lib
```

```
mex myCmexFile.c <matlab>/extern/examples/refbook/fort.c
-I<matlab>/extern/examples/refbook
<matlab>/extern/lib/win32/lcc/libmwlapack.lib
```

For IBM_RS, use

```
mex myCmexFile.c <matlab>/extern/examples/refbook/fort.c
-I<matlab>/extern/examples/refbook -L<matlab>/bin/ibm_rs
-lmwlapack
```

For all other platforms, use

```
mex myCmexFile.c <matlab>/extern/examples/refbook/fort.c
-I<matlab>/extern/examples/refbook
```

Example – Symmetric Indefinite Factorization Using LAPACK

You will find an example C MEX-file that calls two LAPACK functions in the directory `<matlab>/extern/examples/refbook`, where `<matlab>` stands for the MATLAB root directory. There are two versions of this file:

- `utdu_slv.c` – calls functions `zhesvx` and `dsysvx`, and thus is compatible with the PC, HP, and IBM platforms.
- `utdu_slv_.c` – calls functions `zhesvx_` and `dsysvx_`, and thus is compatible with the SGI, LINUX, Solaris, and Alpha platforms.

Calling LAPACK and BLAS Functions from Fortran

You can make calls to the LAPACK and BLAS functions used by MATLAB from your Fortran MEX files. The following is an example program that takes two matrices and multiplies them by calling the LAPACK routine, `dgemm`:

```
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
      integer plhs(*), prhs(*)
      integer nlhs, nrhs
      integer mxcreatedoublematrix, mxgetpr
      integer mxgetm, mxgetn
      integer m, n, p
```

```
      integer A, B, C
      double precision one, zero, ar, br
      character ch1, ch2

      ch1 = 'N'
      ch2 = 'N'
      one = 1.0
      zero = 0.0

      A = mxgetpr(prhs(1))
      B = mxgetpr(prhs(2))
      m = mxgetm(prhs(1))
      p = mxgetn(prhs(1))
      n = mxgetn(prhs(2))

      plhs(1) = mxcreatedoublematrix(m, n, 0.0)
      C = mxgetpr(plhs(1))
      call mxcopyptrtoreal8(A, ar, 1)
      call mxcopyptrtoreal8(B, br, 1)

      call dgemm (ch1, ch2, m, n, p, one, %val(A), m,
     +            %val(B), p, zero, %val(C), m)

      return
      end
```

Building the Fortran MEX-File

The examples in this section show how to compile and link a Fortran MEX file, `myFortranmexFile.F`, on the platforms supported by MATLAB. In each example, the term `<matlab>` stands for the MATLAB root directory.

Building on the PC. On the PC, using Visual Fortran, you will have to link against a library called `libdflapack.lib`:

```
mex  v myFortranMexFile.F
<matlab>/extern/lib/win32/digital/df60/libdflapack.lib
```

Building on IBM_RS. On the IBM_RS, invoke

```
mex myFortranMexFile.F -L<matlab>/bin/ibm_rs -lmwlapack
```

Building on Other UNIX Platforms. On the UNIX platforms (except IBM_RS), you create the MEX file as follows:

```
mex  v myFortranMexFile.F
```


Ex.1049 / Page 413 of 748
Intel Corporation et al

Debugging C Language MEX-Files

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

Note The section entitled, “Troubleshooting” on page 3-31, provides additional information on isolating problems with MEX-files.

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

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

Debugging on UNIX

You 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.

This example shows how to debug `yprime.c` on Solaris using `dbx`, the UNIX debugger.

```
unix> mex -g yprime.c
unix> matlab -Ddbx
<dbx> stop dlopen <matlab>/extern/examples/mex/yprime.mexsol
```

Once the debugger loads MATLAB into memory, you can start it by issuing a `run` command.

```
<dbx> run
```

Now, 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.

```
>> yprime(1,1:4)
<dbx> stop in `yprime.mexsol`mexFunction
```

Note The tick marks used are back ticks (`), not single quotes (').

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 routine. To proceed from the breakpoint, issue a `continue` command to the debugger.

```
<dbx> cont
```

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.

Note For information on debugging on other UNIX platforms, access the MathWorks Technical Support Web site at `http://www.mathworks.com/support`.

Debugging on Windows

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

Microsoft Compiler

If you are using the Microsoft compiler:

1. Start the Microsoft Visual Studio (Version 5 or 6) by typing at the DOS prompt

   msdev *`filename`*.dll

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

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

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

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

Watcom Compiler

If you are using the Watcom compiler:

1 Start the debugger by typing on the DOS command line

```
WDW
```

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

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

Then click **OK**.

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

```
YPRIME
```

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

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

Ex.1049 / Page 416 of 748
Intel Corporation et al

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

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

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

Ex.1049 / Page 417 of 748
Intel Corporation et al

Ex.1049 / Page 418 of 748
Intel Corporation et al

5

Creating Fortran MEX-Files

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

Fortran MEX-Files (p. 5-2)	MEX-file components and required arguments
Examples of Fortran MEX-Files (p. 5-8)	Sample MEX-files that show how to handle all data types
Advanced Topics (p. 5-35)	Help files, linking multiple files, workspace, managing memory
Debugging Fortran Language MEX-Files (p. 5-37)	Debugging MEX-file srouce code from within MATLAB

Fortran MEX-Files

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

MEX-files in Fortran, like their C counterparts, can create any data type supported by MATLAB. You can treat Fortran MEX-files, once compiled, exactly like M-functions.

The Components of a Fortran MEX-File

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

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

The computational and gateway routines may be separate or combined. The following figure, Fortran MEX Cycle, shows how inputs enter an API function, what functions the gateway routine performs, and how output returns to MATLAB.

Ex.1049 / Page 420 of 748
Intel Corporation et al

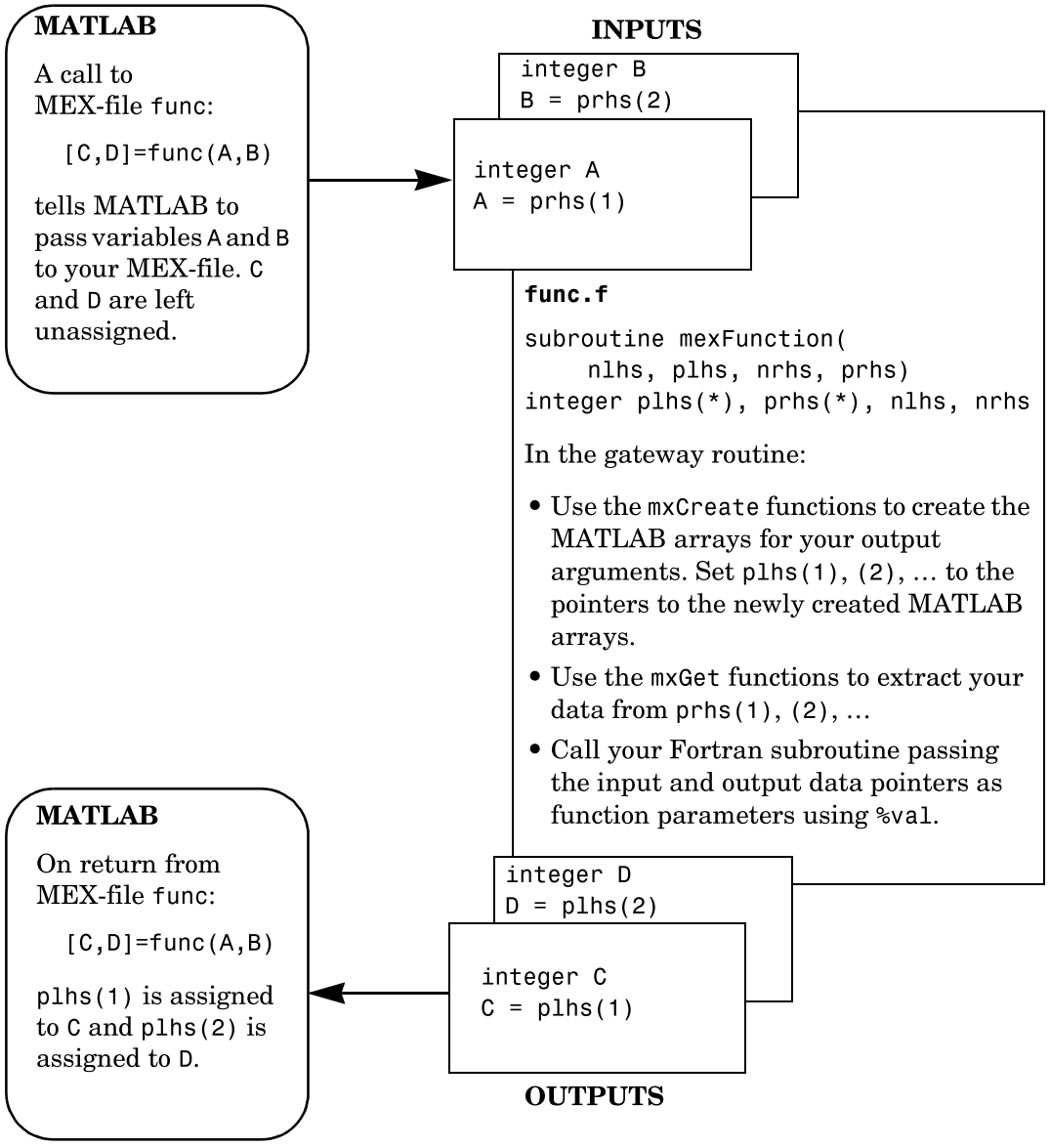


Figure 5-1: Fortran MEX Cycle

The Pointer Concept

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

There are several implications when using pointers in Fortran:

- The `%val` construct.

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

  If your Fortran compiler does not support the `%val` construct, you must use the `mxCopy__` routines (e.g., `mxCopyPtrToReal8`) to access the contents of the pointer. For more information about the `%val` construct and an example, see the section, "The %val Construct" on page 5-7.

- Variable declarations.

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

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

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

Ex.1049 / Page 422 of 748
Intel Corporation et al

The Gateway Routine

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

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

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

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

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

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

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

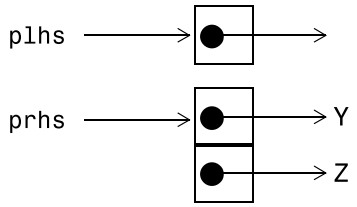
As an example of the gateway routine, consider invoking a MEX-file from the MATLAB workspace with the command

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

the MATLAB interpreter calls `mexFunction` with the arguments

```
nlhs = 1

nrhs = 2
```



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

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

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

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

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

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

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

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

Ex.1049 / Page 424 of 748
Intel Corporation et al

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

The %val Construct

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

```
call yprime(%val(yp), %val(t), %val(y))
```

If your Fortran compiler does not support the `%val` construct, you would replace the call to the computational subroutine with

```
C Copy array pointers to local arrays.
       call mxCopyPtrToReal8(t, tr, 1)
       call mxCopyPtrToReal8(y, yr, 4)
C
C Call the computational subroutine.
       call yprime(ypr, tr, yr)
C
C Copy local array to output array pointer.
       call mxCopyReal8ToPtr(ypr, yp, 4)
```

You must also add the following declaration line to the top of the gateway routine.

```
real*8 ypr(4), tr, yr(4)
```

Note that if you use `mxCopyPtrToReal8` or any of the other `mxCopy__` routines, the size of the arrays declared in the Fortran gateway routine must be greater than or equal to the size of the inputs to the MEX-file coming in from MATLAB. Otherwise `mxCopyPtrToReal8` will not work correctly.

Ex.1049 / Page 425 of 748
Intel Corporation et al

Examples of Fortran MEX-Files

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

- "A First Example — Passing a Scalar"
- "Passing Strings"
- "Passing Arrays of Strings"
- "Passing Matrices"
- "Passing Two or More Inputs or Outputs"
- "Handling Complex Data"
- "Dynamically Allocating Memory"
- "Handling Sparse Matrices"
- "Calling Functions from Fortran MEX-Files"

The MATLAB API provides a set of routines for Fortran that handle double-precision data and strings in MATLAB. For each data type, there is a specific set of functions that you can use for data manipulation.

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

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

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

Ex.1049 / Page 426 of 748
Intel Corporation et al

A First Example — Passing a Scalar

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

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

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

```
C=================================================================
C     timestwo.f
C     Multiply the input argument by 2.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.12 $
C=================================================================

C     Computational subroutine
      subroutine timestwo(y, x)
      real*8 x, y

      y = 2.0 * x
      return
      end


C     The gateway routine
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     64-bit platform.

      integer plhs(*), prhs(*)
      integer mxGetPr, mxCreateDoubleMatrix
      integer x_pr, y_pr
C-----------------------------------------------------------------
```

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

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

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

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

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

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

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

      return
      end
```


Ex.1049 / Page 428 of 748
Intel Corporation et al

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

```
mex timestwo.f
```

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

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

Passing Strings

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

```
C================================================================
C     revord.f
C     Example for illustrating how to copy string data from
C     MATLAB to a Fortran-style string and back again.
C
C     Takes a string and returns a string in reverse order.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.14 $
C================================================================

C     Computational subroutine
      subroutine revord(input_buf, strlen, output_buf)
      character  input_buf(*), output_buf(*)
      integer i, strlen

      do 10 i=1,strlen
         output_buf(i) = input_buf(strlen-i+1)
 10   continue
      return
      end
```

Below is the gateway routine that calls the computational routine.

```
C     The gateway routine
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     platform.

      integer plhs(*), prhs(*)
      integer mxCreateString, mxGetString
C-----------------------------------------------------------------

      integer nlhs, nrhs
      integer  mxGetM, mxGetN, mxIsChar
      integer  status, strlen
      character*100 input_buf, output_buf

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

C     The input  must be a string.
      elseif(mxIsChar(prhs(1)) .ne. 1) then
         call mexErrMsgTxt('Input must be a string.')

C     The input must be a row vector.
      elseif (mxGetM(prhs(1)) .ne. 1) then
         call mexErrMsgTxt('Input must be a row vector.')
      endif

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

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

C     Check if mxGetString is successful.
      if (status .ne. 0) then
```

```
        call mexErrMsgTxt('String length must be less than 100.')
      endif

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

      output_buf = ' '

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

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

      return
      end
```

After checking for the correct number of inputs, this MEX-file gateway routine 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 1, “Importing and Exporting Data.”

For an input string

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

typing

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

produces

```
y =
dlrow olleh
```

Passing Arrays of Strings

Passing arrays of strings adds a slight complication to the example in the previous section, “Passing Strings”. Because MATLAB stores elements of a matrix by column instead of by row, it is essential that the size of the string

array be correctly defined in the Fortran MEX-file. The key point is that the row and column sizes as defined in MATLAB must be reversed in the Fortran MEX-file. Consequently, when returning to MATLAB, the output matrix must be transposed.

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

```
passstr;
```

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

```
C=================================================================
C     passstr.f
C     Example for illustrating how to pass a character matrix
C     from Fortran to MATLAB.
C
C     Passes a string array/character matrix into MATLAB as
C     output arguments rather than placing it directly into the
C     workspace.
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.11 $
C=================================================================

C     The gateway routine
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
C-----------------------------------------------------------------
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     platform.

      integer plhs(*), prhs(*)
      integer p_str, mxCreateString
C-----------------------------------------------------------------
```

```
      integer nlhs, nrhs
      integer i
      character*75 thestring
      character*15 string(5)

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

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

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

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

      return
      end
```

Typing

```
passstr
```

at the MATLAB prompt produces this result

```
ans =

MATLAB
The Scientific
Computing
```

```
Environment
   by TMW, Inc.
```

Passing Matrices

In MATLAB, you can pass matrices into and out of MEX-files written in Fortran. You can manipulate the MATLAB arrays by using `mxGetPr` and `mxGetPi` to assign pointers to the real and imaginary parts of the data stored in the MATLAB arrays. You can create new MATLAB arrays from within your MEX-file by using `mxCreateDoubleMatrix`.

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

```
C=================================================================
C     matsq.f
C
C     Squares the input matrix
C
C     This is a MEX-file for MATLAB.
C     Copyright (c) 1984-2000 The MathWorks, Inc.
C     $Revision: 1.13 $
C=================================================================

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


C     The gateway routine
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)

C-----------------------------------------------------------------
```

```
C     (pointer) Replace integer by integer*8 on the DEC Alpha
C     64-bit platform

      integer plhs(*), prhs(*)
      integer mxCreateDoubleMatrix, mxGetPr
      integer x_pr, y_pr
C-----------------------------------------------------------------

      integer nlhs, nrhs
      integer mxGetM, mxGetN, mxIsNumeric
      integer m, n, size
      real*8  x(1000), y(1000)

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

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

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

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

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

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

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

      return
      end
```

After performing error checking to ensure that the correct number of inputs and outputs was assigned to the gateway subroutine and to verify the input was in fact a numeric matrix, `matsq.f` creates a matrix for the argument returned from the computational subroutine. The input matrix data is then copied to a Fortran matrix by using `mxCopyPtrToReal8`. Now the computational subroutine can be called, and the return argument can then be placed into `y_pr`, the pointer to the output, using `mxCopyReal8ToPtr`.

For a 2-by-3 real matrix

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

typing

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

produces this result

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

Passing Two or More Inputs or Outputs

The `plhs` and `prhs` parameters are vectors that contain pointers to each left-hand side (output) variable and right-hand side (input) variable. Accordingly, `plhs(1)` contains a pointer to the first left-hand side argument, `plhs(2)` contains a pointer to the second left-hand side argument, and so on. Likewise, `prhs(1)` contains a pointer to the first right-hand side argument, `prhs(2)` points to the second, and so on.

Ex.1049 / Page 436 of 748
Intel Corporation et al