Product Support

MEX-files Guide

Section 1: Introduction MEX-files

This technical note provides a general overview of MEX-files and a detailed explanation of the external interface functions that allow you to interface C, C++ or Fortran subroutines to MATLAB. MEX-files are a way to call your custom C, C++ or FORTRAN routines directly from MATLAB as if they were MATLAB built-in functions.

MEX stands for MATLAB Executable. MEX-files are dynamically linked subroutines produced from C, C++ or Fortran source code that, when compiled, can be run from within MATLAB in the same way as MATLAB M-files or built-in functions. The **external interface functions** provide functionality to transfer data between MEX-files and MATLAB, and the ability to call MATLAB functions from C, C++ or Fortran code.

The main reasons to write a MEX-file are:

1. The ability to call large existing C, C++ or FORTRAN routines directly from MATLAB without having to rewrite them as M-files.
2. Speed; you can rewrite bottleneck computations (like for-loops) as a MEX-file for efficiency.

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

Section 2: Getting help

You can learn more about MEX-files from the **MATLAB External Interfaces Guide**. If you already know how to write a MEX-file, you can also use the **External Interfaces Function Reference**.

Note: The MathWorks Technical Support department does not have the resources needed to develop custom code for each specialized application. If, however, a function is not behaving as you think it should, you can **contact Technical Support** for

help.

Section 3: System setup and configuration

MATLAB supports the use of a variety of compilers for building MEX-files. An options file is provided for each supported compiler. You can specify which compiler you want to use. The MathWorks also maintains a **list of compilers supported by MATLAB**.

Once you have verified that you are using a supported C, C++ or FORTRAN compiler, you are ready to configure your system to build MEX-files. In order to do this, run the following command from the MATLAB command prompt:

```
mex -setup
```

When you run this command, a series of questions are asked regarding the location of the C, C++ or Fortran compiler you would like to use to compile your code. After answering these questions, a MEX options file is created that gives MATLAB all of the information it needs to use your compiler during compilation.

Section 4: Testing your system with example MEX-files

Try compiling our sample MEX-file, `yprime.c` found in the <MATLAB>\extern\examples\mex directory.

If you are using C, type the following at the MATLAB prompt to compile the file:

```
mex yprime.c
```

If you are using Fortran, type the following at the MATLAB prompt:

```
mex yprime.f yprimefg.f
```

This creates a MEX-file that can be used at the command prompt like any M-file. If you now type

```
yprime(1,1:4)
```

you should get the following output:

```
ans=
      2.0000
8.9685
4.0000
-1.0947
```

If you do not get this result, or you receive error messages when trying to compile, add a `-v` flag to your compilation command.

```
mex  -v  yprime.c
```

This will produce a lot of output (v is for verbose) that shows the compiling and linking process. This may give more information about why the compilation is failing.

For an example of C++ MEX-File, see **Section 25: C++ MEX-file Overview** below.

Section 5: Troubleshooting system configuration problems

The following resources should offer some insight if you have trouble with any of the above steps.

- The Documentation section on Building MEX Files: **What You Need to Build MEX-Files**
- Solution 1-18TTY: **Why do I get the error "Error: Compile of `yprime.c` failed" when compiling the yprime.c example?**
- Solution 1-175S9: **Why does MATLAB hang or generate an error when I try to run mex -setup or mbuild -setup?**

Section 6: Compiling MEX-files with Microsoft Visual Studio or another IDE

Note that you do not have to compile your MEX-file within an Integrated Development Environment (IDE). Using the MEX utility included with MATLAB may easier and will work just as well; using an IDE is just an alternative.

In general, it is not practical for us to offer complete technical support on the details of using any specific one of the large number of IDE environments our customers use. If you need detailed assistance with the particular settings needed to get your IDE environment to generate code that successfully compiles and runs with our products, we suggest you contact the manufacturer of your IDE to get either information or expert technical assistance in using it.

You can find general Windows custom building information here:

- **Custom Building MEX-Files :: Calling C and Fortran Programs from MATLAB (External Interfaces)**

In addition, the MathWorks Support site contains step-by-step instructions for compiling MEX-files into the following IDEs:

- **MATLAB Documentation** : Microsoft Visual C++

These instructions assume familiarity with your IDE. For additional instructions on how to use your IDE, refer to your IDE documentation.

Section 7: Setting up the MATLAB Add-In for Visual Studio with MATLAB 6.5 (R13)

If you are using MATLAB Compiler 3.0 (R13) and earlier, the MathWorks provides the MATLAB Add-in for the Visual Studio development system that allows you to work within Microsoft Visual C/C++ (MSVC). The MATLAB Add-in for Visual Studio can also be used to build MEX-files in the MSVC environment.

For instructions on setting up the Add-In, see **Solution 1-18L04**.

If you have trouble compiling the MEX-file using the Add-In in MATLAB 6.0 (R12) or earlier, see **Solution 1-18J1I**.

Once you have the Add-In set up, you can use your IDE to compile your MEX-file.

Section 8: The ingredients of a MEX-file

All MEX-files must include four things:

1. `#include mex.h` (C/C++ MEX-files only)
2. `mexFunction` gateway in C/C++ (or SUBROUTINE MEXFUNCTION in Fortran)
3. The mxArray
4. API functions

Section 9: mex.h

Every C/C++ MEX-file must include `mex.h` This is necessary to use the `mx*` and `mex*` routines that are discussed in the **API functions section** of the technical note.

Section 10: mexFunction gateway

The gateway routine to every MEX-file is called `mexFunction`. This is the entry point MATLAB uses to access the DLL.

In C/C++, it is always:

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

In Fortran, it is always:

```
SUBROUTINE MEXFUNCTION( NLHS, PLHS, NRHS, PRHS)
```

Here is what each of the elements mean:

mexFunction	Name of the gateway routine (same for every MEX-file)
nlhs	Number of expected mxArrays (Left Hand Side)
plhs	Array of pointers to expected outputs
nrhs	Number of inputs (Right Hand Side)
prhs	Array of pointers to input data. The input data is read-only and should not be altered by your `mexFunction` .

The variables nrhs and nlhs are the number of variables that MATLAB requested at this instance. They are analogous to **NARGIN and NARGOUT** in MATLAB.

The variables `prhs` and `plhs` are not mxArrays. They are arrays of pointers to mxArrays. So if a function is given three inputs, `prhs` will be an array of three pointers to the mxArrays that contain the data passed in. The variable `prhs` is declared as `const`. This means that the values that are passed into the MEX-file should not be altered. Doing so can cause segmentation violations in MATLAB. The values in plhs are invalid when the MEX-file begins. The mxArrays they point to must be explicitly created before they are used. Compilers will not catch this issue, but it will cause incorrect results or segmentation violations.

Section 11: The mxArray

The mxArray is a special structure that contains MATLAB data. It is the C representation of a MATLAB array. All types of MATLAB arrays (scalars, vectors, matrices, strings, cell arrays, etc.) are mxArrays. For a detailed description of an mxArray, see the **MATLAB External Interfaces Guide**.

The MATLAB language works with only a single object type, the mxArray. All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, and structures are stored as mxArrays. The mxArray declaration corresponds to the internal data structure that MATLAB uses to represent arrays. The MATLAB array is the C language definition of a MATLAB variable. The mxArray structure contains, among other things:

1. The MATLAB variable's name
2. Its dimensions
3. Its type
4. Whether the variable is real or complex

If the variable contains complex numbers as elements, the MATLAB array includes vectors containing the real and imaginary parts. Matrices, or m-by-n arrays, that are not sparse are called *full*. In the case of a full matrix, the mxArray structure contains parameters called `pr` and `pi`. `pr` contains the real part of the matrix data; `pi` contains the imaginary data, if there is any. Both `pr` and `pi` are one-dimensional arrays of double-precision numbers. The elements of the matrix are stored in these arrays column-wise.

An mxArray is declared like any other variable:

```
mxArray *myarray;
```

This creates an mxArray named `myarray`. The values inside myarray are undefined when it's declared, so it should be initialized with an `mx*` routine (such as `mxCreateNumericArray`) before it is used.

It is important to note that the data inside the array is in column major order. Instead of reading a matrix's values across and then down, the values are read down and then across. This is contrary to how C indexing works and means that special care must be taken when accessing the array's elements. To access the data inside of mxArrays, use the API functions (see below).

Section 12: API functions

mx* functions are used to access data inside of mxArrays. They are also used to do memory management and to create and destroy mxArrays. Some useful routines are:

Array creation	mxCreateNumericArray, mxCreateCellArray, mxCreateCharArray
Array access	mxGetPr, mxGetPi, mxGetM, mxGetData, mxGetCell
Array modification	mxSetPr, mxSetPi, mxSetData, mxSetField

Memory management	mxMalloc, mxCalloc, mxFree, mexMakeMemoryPersistent, mexAtExit, mxDestroyArray, memcpy

Remember that `mxGetPr` and `mxGetPi` return pointers to their data. To change the values in the array, it is necessary to directly change the value in the array pointed at, or use a function like `memcpy` from the C Standard Library.

```
#include <string.h>
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ])
{
int j;
double *output;
double data[] = {1.0, 2.1, 3.0};


/* Create the array */
plhs[0] = mxCreateDoubleMatrix(1,3,mxREAL);
output = mxGetPr(plhs[0]);


/* Populate the output */
memcpy(output, data, 3*sizeof(double));

/* Alternate method to populate the output, one element at a time */
/* for (j = 0; j < 3; j++) */
/* { */
/*   output[j] = data[j]; */
/* } */

}
```

The API functions `mxCalloc` and `mxFree` etc. should be used instead of their Standard C counterparts because the `mx*` routines let MATLAB manage the memory and perform initialization and cleanup.

On the PC there is no concept of `stdin`, `stdout` and `stderr`, so it is important to use MATLAB's functions such as `mexPrintf` and `mexError`. A full list of `mx*` routines with complete descriptions can be found in the **MATLAB External/API Reference Guide**.

`mex*` functions perform operations back in MATLAB.

Some useful routines are:

mexFunction	Entry point to C MEX-file
mexErrMsgTxt	Issue error message and return to MATLAB
mexEvalString	Execute MATLAB command in caller's workspace
mexCallMATLAB	Call MATLAB function or user-defined M-file or MEX-file
mexGetArray	Get copy of variable from another workspace
mexPrintf	ANSI C printf -style output routine
mexWarnMsgTxt	Issue warning message

A full list of `mex*` routines with complete descriptions can be found in the **MATLAB External/API Reference Guide**.

The MEX API provides several functions that allow you to determine various states of an array. These functions are used to check the input to the MEX-file, to make sure it is what's expected. All of these functions begin with the `mxIs` prefix. In some cases it may be desirable to use the specific `mxIs` function for a specific datatype. However, it is much easier, in general, to use `mxIsClass` to perform this operation.

In order to prevent passing inputs that are the incorrect type, use the `mxIsClass` function extensively at the beginning of your MEX-file. For example, suppose `prhs[0]` is supposed to be a regular, full, real-valued array. To prevent passing your function a sparse matrix, a string matrix, or a complex matrix, use code similar to the following:

```
if ( mxIsChar(prhs[0]) || mxIsClass(prhs[0], "sparse") ||
mxIsComplex(prhs[0]) )
mexErrMsgTxt("first input must be real, full, and nonstring");
```

Putting these checks in your code will make your MEX-file more robust. Instead of causing a segmentation fault when passed incorrect arguments, it will instead produce an error message and return you to the MATLAB Command Prompt.

Section 13: Example: Writing a "Hello World" MEX-file

In this first example, we will create a MEX-file (hello.c) that prints "hello world" to the screen. We will then build and run the MEX-file from MATLAB.

1. As described in the ingredients section, every MEX-file includes `mex.h`. Thus, your MEX source should start like this:

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

2. Every MEX-file has the `mexFunction` entry point. The souce now becomes

   ```
   #include "mex.h"
   void mexFunction(int nlhs, mxArray *plhs[],
   ```

```
    int nrhs, const mxArray *prhs[]) {
```

3. Add an API function to make the MEX-file do something. The final version of the souce becomes:

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[]) {
  mexPrintf("Hello, world!\n");
}
```

   Your first MEX-file is complete. Save it as `hello.c`.

4. The next step is to tell MATLAB which compiler you want to use to build the MEX-file. You do this with the `mex -setup` command. You can choose the LCC compiler, the C compiler included with MATLAB. This is what it looks like from the MATLAB command prompt:

```
mex -setup
```

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

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

Select a compiler:
[1] Lcc C version 2.4.1 in D:\APPLIC~1\MATLAB\R2006b\sys\lcc
[2] Microsoft Visual C/C++ version 8.0 in D:\Applications\Microsoft Visual Studio 8
[3] Microsoft Visual C/C++ version 7.1 in D:\Applications\Microsoft Visual Studio .NET 2003

[0] None

Compiler: 1

Please verify your choices:

Compiler: Lcc C 2.4.1
Location: D:\APPLIC~1\MATLAB\R2006b\sys\lcc

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

Trying to update options file:
   C:\WINNT\Profiles\username\Application Data\...
   MathWorks\MATLAB\R2006b\mexopts.bat
From template:
   D:\APPLIC~1\MATLAB\R2006b\bin\win32\mexopts\lccopts.bat

Done . . .
```

5. Now you are ready to compile and link the MEX-file. You can do this with the following command:

```
mex hello.c
```

   Notice that `hello.mexw32` (the MATLAB callable MEX-file for 32-bit Windows) is created in the current directory; MEX-file extensions vary by platform.

6. You can now call the MEX-file like any other M-file by typing its name at the MATLAB command prompt.

```
hello
```

```
Hello, world!
```

Section 14: Example: Using API routines to work with mxArrays

In the example below, we will create a MEX-file that takes any number of inputs and creates and equal number outputs. The output values will be twice the input values.

1. The first job of the MEX-file is to create mxArrays to hold the output data. Each output will be the same size as its corresponding input. This is done using `mxCreateDoubleMatrix` (creating a matrix to hold doubles), `mxGetM` (the number of rows the output should be), and `mxGetN` (the number of columns the output should be).
2. After the output mxArray is created, the only things left to do is to multiply the input by two, and to put that value into the output array. This is done with `mxGetPr` (get pointers to the input data and the newly-created output mxArray).
3. The source code for this example is

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
 int i, j, m, n;
 double *data1, *data2;
 if (nrhs != nlhs)
 mexErrMsgTxt("The number of input and output arguments must be the same.");


 for (i = 0; i < nrhs; i++)
   {
    /* Find the dimensions of the data */
    m = mxGetM(prhs[i]);
```

```
        n = mxGetN(prhs[i]);


        /* Create an mxArray for the output data */
        plhs[i] = mxCreateDoubleMatrix(m, n, mxREAL);


        /* Retrieve the input data */
        data1 = mxGetPr(prhs[i]);


        /* Create a pointer to the output data */
        data2 = mxGetPr(plhs[i]);


         /* Put data in the output array */
        for (j = 0; j < m*n; j++)
        {
        data2[j] = 2 * data1[j];
        }
       }

    }
```

Save the source as `timestwo.c`

4. The MEX-file can now be compiled.

```
mex -setup %choose your C compiler
    %(LCC is fine for this example)
mex timestwo.c
```

5. Now the MEX-file can be called from MATLAB like any other M-file. For example,

```
  [a,b]=timestwo([1 2 3 4; 5 6 7 8], 8)
```

```
a =
                 2   4   6   8
                 10  12  14  16
```

```
b =
                       16
```

Section 15: Example: Checking inputs and outputs via a MEX-file

```
/* The following is a very basic
MEX-file that checks to make sure
that its input is a scalar.
Note that it is written in ANSI C. */


 #include "mex.h"


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


{
int m, n;
double x, *y;


/* check: only one input and one output argument */
if (nrhs !=1)
  mexErrMsgTxt("Must have one input argument");
if (nlhs !=1)
  mexErrMsgTxt("Must have one output argument");


/* prevent you from passing a sparse matrix,
a string matrix, or a complex array. mxIsComplex
is used to determine if there is an imaginary
part of the mxArray. mxIsClass is used to determine
if the mxArray belongs to a particular class */


if ( mxIsComplex(prhs[0])|| mxIsClass(prhs[0],
"sparse") || mxIsChar(prhs[0]) )
  mexErrMsgTxt("Input must be real, full,
```

```
  and nonstring");
 }
```

The function `mexErrMsgTxt` works like the MATLAB function `ERROR`. When called, it exits the MEX-file and reports an error message specified in the input string.

This MEX-file example first checks to make sure that the function was called correctly, with the correct number of inputs and outputs. It then verifies that the input is a scalar. Finally,using the `mxIs*` functions, it verifies that the input is a full nonsparse array.

Section 16: Example: Passing arrays between MEX-files and MATLAB

Usually, arrays are passed to MATLAB via the right-hand side (rhs) and the left-hand side (lhs) method. This means that variables are passed into and out of a function by being included as arguments to the function. Sometimes, there are cases in which you may need to violate this standard. Some examples are:

1. You have more than 50 input or output variables (MATLAB has a limit of 50)
2. You want to modify many variables using a function, and you don't want to have to type

```
[a,b,c,d,e,f,g,h,...] = func(a,b,c,d,e,f,g,h,...)
```

Unlike function M-files, MEX functions have the unique ability to get matrices from the workspace of the calling function, without having the matrix passed in via the prhs structure. The calling function is the M-file function from which the MEX-file is called. When a MEX-file is called from an M-file script or the command line, the calling function's workspace is the main MATLAB workspace. For example, assume you call the following function.

```
function thefun

x=5; y=0;

themexfun(x);
```

Even though `themexfun` is only passed the variable x, it can still gain access to the variable y. The following code fragment shows one way this can be done from inside the MEX-file.

```
        const mxArray *array_ptr;
        array_ptr = mexGetVariable("y", "caller");
```

When this code is executed, a copy of the variable y is made, and array_ptr is assigned to point to it. The following examples describe the three `mexGet*` functions that can access an array in more detail. The examples use these variable declarations:

```
const mxArray *array_ptr;
int m,n, errcode;
double *pr, *pi;
char *name;
```

Example of using `mexGetVariable`

`array_ptr=mexGetVariable(name, workspace)` makes a copy of the matrix whose name is specified from the base workspace. It returns a pointer to the copy, or NULL if the matrix doesn't exist.

```
const mxArray *mymatrix;
if ((mymatrix = mexGetVariable("a", "base"))==NULL)
  mexErrMsgTxt("Variable 'a' not in workspace.");
else {
  <do stuff with mymatrix>
}
```

Example of using `mexGetVariablePtr`

```
array_ptr=mexGetVariablePtr(name, workspace)
```

is similar to `mexGetVariable`, but instead of making a copy of the array, a pointer to the original array is returned. The only thing you should do with `array_ptr` is examine the array's data and characteristics. If you need to change data, call `mexGetVariable`:

```
const mxArray *myarray;


if ((myarray = mexGetVariablePtr("a", "base"))==NULL)
  mexErrMsgTxt("Variable 'a' not in workspace.");
else {
  <do stuff with mymatrix>
}
```

In general, be careful when using any of the `mexGet*` or `mexPut*` functions. These functions, especially the `mexPut*` functions, are likely to have strange side effects, which at best create strange variables in your workspace, and at worst write over your data without you knowing about it.

Section 17: Example: Calling MATLAB functions from MEX-files

There are two functions that allow you to call other MATLAB functions.

1. **mexEvalString**
2. **mexCallMATLAB**

Using `mexEvalString`

`errcode=mexEvalString(str)` is similar to MATLAB's **`EVAL`** function. It evaluates its string input in the calling function's workspace, in the same manner as if it had been entered at the MATLAB command line.

For example,

```
mexEvalString("p=plot(1:10);");
```

would generate a plot of 1 to 10 in the current figure window. Note that since the command is evaluated in the calling function's workspace, the variable p is stored in the calling function's workspace. Thus, if the MEX-file were to be called from within an M-file function, p would be stored in that function's workspace, not the main workspace.

Although `mexEvalString` is easy to code, the string passed to `mexEvalString` is evaluated by MATLAB, therefore, the MATLAB parser has to be called. Calling another MEX-file via `mexEvalString` is not very efficient since the MATLAB parser is called. Also, all variables created by the `mexEvalString` command are stored in the workspace of the function text called by the MEX-file, which may need to be retrieved with the `mexGet*` functions. A more efficient way to accomplish the same goal is to use the `mexCallMATLAB` function (assuming you are calling a function and not a script).

Using `mexCallMATLAB`

Other MATLAB functions including built-in functions, MEX-file and M-file functions (not M-file scripts) can be called from within a MEX-file using the `mexCallMATLAB` function.

This function accepts five inputs. The first four are structured as integers, exactly the same as the inputs to `mexFunction`: **nlhs, plhs, nrhs,** and **prhs**.

`plhs` and `prhs` are pointers to arrays of mxArrays. You set up nrhs, `prhs`, and nlhs to contain what you want passed to the MATLAB function that you are calling. The function returns any output data in `plhs`. Thus, it is as if your MEX-file is acting as the MATLAB parser, arranging the data in the correct data structure and passing it to the function. The fifth argument is a character string containing the name of the MATLAB function to be called. The following is an example of this that calculates the determinant of a matrix and prints it out. The example assumes the matrix pointer array_ptr has already been initialized to contain valid data.

```
int nlhs1, nrhs1; /*chances are plhs and prhs are
already used by mexFunction */
mxArray *plhs1[1], *prhs1[1];


nlhs1 = 1;  /* One output requested from the
  'det' function */
nrhs1 = 1;  /* One input passed to the 'det'
  function */
prhs1[0] = array_ptr; /* Set up input argument to 'det' */


mexCallMATLAB(nlhs1,plhs1,nrhs1,prhs1,"det");
mexPrintf("Det(array_ptr) == %g",mxGetScalar(plhs1[0]));
```

Section 18: Additional MEX examples

The **MATLAB External Interfaces Guide** has several example MEX-files dealing with different types of data including scalars, strings, structures, cell arrays, and sparse arrays.

If you are writing FORTRAN MEX-files, similar examples can be found **here**.

Section 19: Custom options files

The `mex` script has a set of switches (also called options) that you can use to modify the link and compile stages. For a list with descriptions of switches available, type:

```
mex -help
```

at the MATLAB command prompt.

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

Depending on your platform, the `mex` script looks for an options file in the following location:

On UNIX:

1. The current directory
2. The user profile directory (returned by the PREFDIR function)
3. The directory specified by `[matlabroot '/bin']`

On Windows:

1. The current directory
2. The user profile directory (returned by the PREFDIR function)
3. The directory specified by `[matlabroot '\bin\win32\mexopts']` for 32-bit Windows, or `[matlabroot '\bin\win64\mexopts']` for 64-bit Windows

Section 20: Linking multiple files

It is possible to combine several object files and 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 Windows:

```
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.mexw32
```

with the appropriate MEX-file extension for your platform (here "mexw32" for 32-bit Windows). The base 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, 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.

Section 21: General debugging steps

In general, these are the steps to debug a MEX-file:

1. Something bad happens (such as a segmentation fault)
2. Compile with `-g`. For example

```
mex -g yourmexfile.c
```

3. Invoke MATLAB through the debugger
4. Turn MEX debugging on (UNIX only)
5. Trace your code until it reaches the issue
6. Use the debugger to investigate
7. Fix the problem
8. Compile without `-g`
9. Repeat as necessary

Section 22: Debugging on Windows

You can find general Windows debugging information here:

- **Debugging on Windows :: Creating C Language MEX-Files (External Interfaces)**
- **Debugging on Windows :: Creating Fortran MEX-Files (External Interfaces)**

In addition, the MathWorks Support site contains step-by-step instructions for loading MEX-files into the following debuggers:

- **MATLAB Documentation** : Microsoft Visual Studio 2005
- Solution **1-3KK6RK** : Microsoft Visual C++ 2005
- Solution **1-1BUHC** : Microsoft Visual C++ .NET 2003
- Solution **1-1Y1FEG** : Microsoft Visual C/C++ 6.0
- Solution **1-2Y9F7M** : Borland C++Builder 6
- Solution **1-32VRZ1** : Intel, Compaq, or Digital Visual Fortran for Fortran MEX-files
- Note: The LCC compiler shipped with MATLAB does not include a debugger. You can use MEXPRINTF to print status messages to the screen, but will not have access to breakpoints and controlled execution as with other compilers.

These instructions assume familiarity with your debugger. For instructions on how to use your debugger once the MEX-file is loaded, refer to your debugger documentation.

Section 23: Debugging on Linux/UNIX/Mac

You can find general Linux/UNIX/Mac debugging information here:

- **Debugging on UNIX :: Creating C Language MEX-Files (External Interfaces)**
- **Debugging on UNIX :: Creating Fortran MEX-Files (External Interfaces)**

In addition, the MathWorks Support site contains step-by-step instructions for the following debuggers:

- **MATLAB Documentation** : GDB for C
- **MATLAB Documentation** : GDB for Fortran
- Solution **1-17Z0R** : GDB and DBX on numerous UNIX platforms. This includes platforms supported by past versions of MATLAB.
- Solution **1-19DVQ** : GDB on Solaris
- Solution **1-22BGKF** : Xcode on Mac
- Solution **1-35LZ0B** : GDB for Fortran MEX-files on Linux/UNIX/Mac

These instructions assume familiarity with your debugger. For instructions on how to use your debugger once the MEX-file is loaded, refer to your debugger documentation.

Useful GDB commands

file	Read a source file in
step	Move forward one line in debugging
where, whereami	Show the call stack, show the current location
list	Show the current line and following source file contents
print	Display the value of a variable
stop in, stop at	Stop in a particular function, stop at a particular line
help	Show the help
what	Tell what a thing is, give the function prototype, etc.

Section 24: Using other debuggers

If using another debugger, start MATLAB with the -D flag, specifying your debugger:

`matlab -Dgdb`	The GNU debugger

`matlab -Dddd`	A graphical front end to GDB
`matlab -D"workshop -D <path to matlab.exe>"`	The Sun Workshop debugger

Section 25: C++ MEX-file Overview

MEX-files can be written in C++. See `mexcpp.cpp` in the `$MATLABROOT/extern/examples/mex` directory (where `$MATLABROOT` is the MATLAB root directory on your machine, as returned by typing `matlabroot` at the MATLAB Command Prompt.)

Section 26: Tips for C++ MEX-files

Here are some tips to keep in mind if you decide to use C++ in your MEX-file.

1. For an example MEX-file, see:

   ```
   <MATLAB>/extern/examples/mex/mexcpp.cpp
   ```

   The extension .cpp is unambiguous and generally recognized by C++ compilers. Other possible extensions include .C, .cc, and .cxx.
2. Using cout will not work as expected in C++ MEX-files. This is because cout is expecting to use a display that is not MATLAB. To workaround this problem, use **`mexPrintf`** instead.
3. If you run your MEX-file in MATLAB and you do not receive the expected output, make sure that you have a C++ `flush()` function call in your program.

Section 27: Compiling C++ MEX-files

Use

```
mex -setup
```

to select a C++ compiler at the MATLAB command prompt. The LCC compiler shipped with MATLAB is a C-only compiler, and cannot be used with C++. Next, invoke MEX as:

```
mex mexcpp.cpp
```

Special instructions for MATLAB 6.1 (R12.1) and earlier on UNIX: In order to compile this example, you should copy the options file for C++ MEX-files (<MATLAB>/bin/cxxopts.sh) to your current directory. Then, invoke MEX as:

```
mex mexcpp.cpp -f cxxopts.sh
```

This is not required for MATLAB 6.5 (R13) and later on UNIX, where a C++ compiler can be selected using `mex -setup` as shown above.

Section 28: If Linking fails

1. Retry compiling with

   ```
   yprime.c
   ```

2. Try running

   ```
   mex -setup
   ```

   again
3. Check that you are using a supported compiler. A list of supported compilers can be found in the **Supported and Compatible Compiler List**.
4. Verify that the code is correct C code
5. If you are getting linking, unresolved external, or undefined symbols errors, follow these steps:
   a. Find the name of the symbol
   b. It is a MathWorks symbol? Does it begin with "_mx", "_mex", "_eng","_mat", "_mlf", or "_mcl"?
   c. If it is a MathWorks symbol, ensure you are linking aganist the correct libraries

Section 29: If loading fails

1. Dependency Walker is a free tool that can help identify missing DLL-files, one common cause of MEX-files that fail to load on Windows. Solution **1-2RQL4L** explains how to use Dependency Walker to diagnose this issue.
2. Ensure the MEX-file is a 32- or 64-bit shared library as appropriate for your platform. In particular, Windows MEX-files cannot depend on 16-bit libraries.
3. Ensure <MATLAB>/extern/lib/$ARCH is included in $LD_LIBRARY_PATH
4. Messages from `ld.so` usually indicate a problem with the library path
5. On Windows, ensure that the directories of all necessary external DLL-files are in the PATH variable

Section 30: If running fails - segmentation violations:

Memory is grouped into blocks, or segments. A key function of every operating system is to keep track of which processes own the various memory segments. A process can only access memory which it owns. This prevents critical data from being overwritten and maintains security.

A segmentation violation occurs when a process attempts to access memory which it does not own. This typically happens when a user tries to write past the end of an array, access dynamically allocated data that has previously been freed, or de-reference a NULL pointer.

If you encounter a problem like this, debug the MEX-file using the steps in the **debugging section of this technical note**.