

how do i...?

1605 MEX-files Guide

Revision: 1.0

Last Date Modified: 04-August-2003

Introduction

1. [Introduction to MEX-files](#)
2. [Getting help](#)

Compiling MEX-files

3. [System setup and configuration](#)
4. [Testing your system with example MEX-files](#)
5. [Troubleshooting system configuration problems](#)
6. [Compiling MEX-files with the Microsoft Visual C++ IDE](#)
7. [Setting up the MATLAB Add-In for Visual Studio](#)

MEX-file components

8. [The ingredients of a MEX-file](#)
9. [mex.h](#)
10. [mexFunction gateway](#)
11. [The mxArray](#)
12. [API functions](#)

MEX-file examples

13. [Writing a "Hello World" MEX-file](#)
14. [Using API routines to work with mxArrays](#)
15. [Checking inputs and outputs via a MEX-file](#)
16. [Passing arrays between MEX-files and MATLAB](#)
17. [Calling MATLAB functions from MEX-files](#)
18. [Additional MEX examples](#)

Advanced MEX options

19. [Custom options files](#)
20. [Linking multiple files](#)

Debugging MEX-files

21. [General debugging steps](#)
22. [Debugging with the Microsoft Developer's Studio](#)
23. [Debugging with DBX in UNIX](#)
24. [Using other debuggers](#)

C++ MEX-files

25. [WARNING](#)
26. [Tips for C++ MEX-files](#)
27. [Compiling C++ MEX-files](#)

Troubleshooting MEX problems

28. [If linking fails](#)
 29. [If loading fails](#)
 30. [If running fails - segmentation violations](#)
-

Section 1: Introduction MEX-files

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

MEX stands for MATLAB Executable. MEX-files are dynamically linked subroutines produced from C or Fortran source code, can be run from within MATLAB in the same way as MATLAB M-files or built-in functions. The [external interface](#) functionality to transfer data between MEX-files and MATLAB, and the ability to call MATLAB functions from C or Fortran.

The main reasons to write a MEX-file are:

1. The ability to call large existing C or FORTRAN routines directly from MATLAB without having to rewrite them.
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 tedious level programming in compiled languages like C or Fortran. In general, most programming should be done in MATLAB 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, use the [External Interfaces Function Reference](#).

NOTE: The MathWorks Technical Support department does not have the resources needed to develop custom code for your 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 to 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 or FORTRAN compiler, you are ready to configure your system. In order to do this, please 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 or Fortran compiler you want 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 find 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, please type the following at the MATLAB prompt to compile the file:

```
mex yprime.c
```

If you are using Fortran, please 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, please add a -v flag to your compiler:

```
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 info if the compilation is failing.

Section 5: Troubleshooting system configuration problems

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

- Technical Note 1621: [What should I do before trying to compile?](#)
- Solution 29885: [Troubleshooting problems with mex -v yprime.c](#)
- Solution 29876: [Why do I get the error "Error: Compile of 'yprime.c' failed" when compiling the yprime.c example?](#)
- Solution 22688: [Why does MATLAB hang or generate an error when I try to run mex -setup or mbuild -setup?](#)

Section 6: Compiling MEX-files with the Microsoft Visual C++ IDE

Please note that you do not have to compile your MEX-file within an IDE. Using the MEX utility included with MATLAB just as well; using MSVC is just an alternative. In addition, you will need knowledge of MSVC to do this.

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

For the details on project settings to compile MEX-files, please see the [MATLAB External Interfaces/API Reference](#). You can also refer to [Solution 26470](#).

Section 7: Setting up the MATLAB Add-In for Visual Studio

The MathWorks provides a [MATLAB Add-in for the Visual Studio](#) development system that allows you to work easily with C/C++ (MSVC). The MATLAB Add-in for Visual Studio greatly simplifies using MEX-files in the MSVC environment.

For instructions on setting up the Add-In, please see [Solution 29041](#).

If you have trouble compiling the MEX-file, please see [Solution 28875](#).

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:

<http://www.mathworks.com/support/tech-notes/1600/1605.shtml>

3/4/2004

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 [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:

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

The variables `nrhs` and `nlhs` are the number of variables that MATLAB requested at this instance. They are analogous to `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, it should return 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 data that are passed into the MEX-file should not be altered. Doing so can cause segmentation violations in MATLAB. The `prhs` array is invalid when the MEX-file begins. The `mxArrays` they point to must be explicitly created before they are used. Compiler error is a problem, 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 (scalars, vectors, matrices, strings, cell arrays, etc.) are `mxArrays`. For a detailed description of an `mxArray`, please see [External Interfaces Guide](#).

The MATLAB language works with only a single object type, the `mxArray`. All MATLAB variables, including scalars, vectors, cell arrays, and structures are stored as `mxArrays`. The `mxArray` declaration corresponds to the internal data structure that represents arrays. The MATLAB array is the C language definition of a MATLAB variable. The `mxArray` structure contains the following 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.

or m-by-n arrays, that are not sparse are called full. In the case of a full matrix, the `mxArray` structure contains parameter `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 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 by a `mx*` routine (such as `mxCreateNumericArray`) before it is used.

It is important to note that the data inside the array is in row major order. Instead of reading a matrix's values across and then down, 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`, please 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 `mxArrays`. Some useful routines are:

Array creation	<code>mxCreateNumericArray</code> , <code>mxCreateCellArray</code> , <code>mxCreateCharArray</code>
Array access	<code>mxGetPr</code> , <code>mxGetPi</code> , <code>mxGetM</code> , <code>mxGetData</code> , <code>mxGetCell</code>
Array modification	<code>mxSetPr</code> , <code>mxSetPi</code> , <code>mxSetData</code> , <code>mxSetField</code>
Memory management	<code>mxMalloc</code> , <code>mxCalloc</code> , <code>mxFree</code> , <code>mexMakeMemoryPersistent</code> , <code>mexAtExit</code> , <code>mxImaginary</code> , <code>memcpy</code>

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

```
int i;
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 */
for (j = 0; j < 3; j++)
    memcpy(output + j, data + j, sizeof(double));
output[j] = data[j]
```

The API functions `mxCalloc` and `mxFree` etc. should be used instead of their Standard C counterparts because the MATLAB runtime manages the memory and performs 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 `mexWarnMsgTxt`. 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:

<code>mexFunction</code>	Entry point to C MEX-file
<code>mexErrMsgTxt</code>	Issue error message and return to MATLAB
<code>mexEvalString</code>	Execute MATLAB command in caller's workspace
<code>mexCallMATLAB</code>	Call MATLAB function or user-defined M-file or MEX-file
<code>mexGetArray</code>	Get copy of variable from another workspace
<code>mexPrintf</code>	ANSI C <code>printf</code> -style output routine
<code>mexWarnMsgTxt</code>	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 in the MEX-file, to make sure it is what's expected. All of these functions begin with the `mxIs` prefix. In some cases it is necessary to use the specific `mxIs` function for a specific datatype. However, it is much easier, in general, to use `mxIsClass` to perform these checks.

In order to prevent passing inputs that are the incorrect type, use the `mxIsClass` function extensively at the beginning of your function. For example, suppose `prhs[0]` is supposed to be a regular, full, real-valued array. To prevent passing your function a sparse 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 prevents your MEX-file from crashing for no apparent reason when you accidentally pass in the wrong type of data.

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

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

2. Every MEX-file has the `mexFunction` entry point. The source 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 source becomes:

```
#include "mex.h"

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

Our first MEX-file is complete. Please 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` 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 in D:\MATLAB6P1\sys\lcc
[2] Microsoft Visual C/C++ version 6.0 in
    D:\Applications\Microsoft Visual Studio
[0] None

Compiler: 1

Please verify your choices:
Compiler: Lcc C 2.4
```

```
Location: D:\MATLAB6P1\sys\lcc
Are these correct?([y]/n): y
```

The default options file:

```
"C:\WINNT\Profiles\username.MATHWORKS\Application Data\...
  MathWorks\MATLAB\R12\mexopts.bat" is being updated...
from D:\MATLAB6P1\BIN\WIN32\mexopts\lccopts.bat...
```

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.dll` (the MATLAB callable dll) is created in the current directory.

```
>> mex hello.c
```

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 mxArray

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

1. The first job of the MEX-file is to create mxArray to hold the output data. Each output will be the same size as the input. This is done using `mxCreateDoubleMatrix` (creating a matrix to hold doubles), `mxGetM` (the number of rows) 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. This is done with `mxGetPr` (get a pointer to the real part of the input data) and `mxMalloc` (the MEX version of the C function `malloc`).

Note: Anywhere where you would use `malloc/free` in a C program, you should use `mxMalloc` and `mxFree`. To get more help on these or any other function, please see the [API Function Reference](#).

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 dimension of the data */
        m = mxGetM(prhs[i]);
        n = mxGetN(prhs[i]);
```

```

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

/* Get the data passed in */
data1 = mxGetPr(prhs[i]);

/* Create an array for the output's data */
data2 = (double *) mxMalloc(m*n * sizeof(double));

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

/* Assign the data array to the output array */
mxSetPr(plhs[i], data2);
}
}

```

Please save the source as `timestwo.c`

4. The MEX-file can now be compiled.

```

mex -setup %choose your C compiler
        %(LCC is fine for the 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 input 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 = mexGetArray("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 `mexGetArray`

`array_ptr=mexGetArray(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 = mexGetMatrix("a", "base"))==NULL)
    mexErrMsgTxt("Variable 'a' not in workspace.");

else {
    <do stuff with mymatrix>
}
```

Example of using `mexGetArrayPtr`

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

is similar to `mexGetArray`, 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 `mxGetArray`:

```
const mxArray *myarray;

if ((myarray = mexGetMatrixPtr("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,

<http://www.mathworks.com/support/tech-notes/1600/1605.shtml>

```
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're 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];

prhs1 = 1; /* One output requested from the
           'det' function */
prhs1 = 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, please 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. `$HOME/.matlab/R12`
3. `<matlab>/bin`

On a PC:

1. The current directory
2. The user profile directory. See the section below for more information on this directory.
3. `<matlab>\bin\win32\mexopts`

The user profile directory

The Windows user profile directory contains user-specific information such as desktop appearance, recently used files, and Start menu items.

The `mex` and `mbuild` utilities store their respective options files that are created during the `-setup` process, `mexopts.bat` and `compopts.bat` in a subdirectory of your user profile directory, named Application Data\MathWorks\MATLAB.

Under Windows NT and Windows 95/98 with user profiles enabled, your user profile directory is `%windir%\Profiles\username`. Under Windows 95/98 with user profiles disabled, your user profile directory is `%windir%`. Under Windows 95/98, you can choose whether or not user profiles are enabled by using the Passwords control panel.

Under Unix, MATLAB information is stored in `$HOME/.matlab`

Running `mex -setup` creates an options file in the user's home/profile directory for `mex` to use. In order to use a custom options file, you can either copy your file to one of the locations listed above or you can use the `-f` option. For example:

```
mex -f /home/username/mexoptions/myopts.sh ...
    <path to your options file>
```

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 a 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, 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 in debugging:

1. Something bad happens
2. Compile with `-g`. For example


```
mex -g yourmexfile.c
```
3. Invoke MATLAB through the debugger
4. Turn MEX debugging on (UNIX/VMS only)
5. Encounter badness
6. Use the debugger
7. Fix the problem
8. Compile without `-g`
9. Repeat as necessary

Section 22: Debugging with the Microsoft Developer's Studio

1. Run


```
vcvars32.bat
```

from a DOS prompt to setup the compiler environment
2. Run the debugger:


```
msdev filename.dll
```
3. Tell MSVC to use MATLAB: Project -> Settings -> Debug tab. Change "Executable for Debug Session" to full path to `matlab.exe`
4. Add the source file(s): File -> Open
5. Set some breakpoints in the code: Navigate to the line and press F9
6. Start debugging: Build -> Start Debugging -> Go
7. Run the MEX-file in MATLAB
8. Step through the program

NOTE: Your debugger window might not look exactly like this, but the components should all be there. The major components are described above. Also, the `vcvars32.bat` file doesn't need to be run if the compiler is installed on your machine. If it is not or if multiple versions of the same compiler are installed, you should run this.

You may get an error message,

`matlab.exe` contains no debugging information.

The MathWorks does not distribute a debuggable version of MATLAB, so this error is to be expected and can safely be ignored.

Once the breakpoint is encountered you should use one of the "step" functions on the left-hand side above. "Step Into" moves you into the function call on the current line. "Step out" finishes the current function and moves back into the calling code. "Step over" unlike "step into" executes the current line but does not move into the function that is called. "Run to cursor" allows you to skip big sections of code.

The stack indicates all of the functions that were called to get to the current location. You can select a different function from the stack to see how the current function was called and to access the calling function's variables. The watch window and current value window allow you to inspect the values in the function as they change.

Section 23: Debugging with DBX in UNIX

This sections assumes some knowledge about DBX.

1. Start up MATLAB with the debugger:

```
matlab -Ddbx
```

This also starts the debugger.

2. Continue loading MATLAB by typing

```
run
```

at the debugger prompt.

3. After MATLAB starts, enable debugging:

```
dbmex on
```

4. Call the MEX-file as you normally would; this drops you into the debugger.
5. Set breakpoints and issue the `continue` command.
6. Once the breakpoint is hit, use the normal debugger routines.

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

You can invoke MATLAB with various flags for various debuggers.

<code>matlab -Dgdb</code>	The GNU debugger
---------------------------	------------------

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

Section 25: C++ MEX-files - WARNING

MEX-files are only officially supported in MATLAB if they are written in C or FORTRAN. C++ MEX-files are not officially supported. With some work and tweaking on your own, you may be able to get a C++ MEX-file to work as well.

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. The `mexFunction` definition, along with included headers should be wrapped with `extern "C"` so that it has "C linkage" for the benefit of MATLAB. Within the body of the `mexFunction` definition, full C++ syntax is allowed. For an example MEX-file, please see:

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

The extension `.cpp` was chosen rather arbitrarily, but mainly for being unambiguous and generally recognized by C++ compilers. The `mex` script should recognize any other common C++ source file extension, such as `.C`, `.cc`, or `.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 workaroud this problem, please use `mexPrintf` instead.
3. The MathWorks has had the best luck compiling C++ MEX-files with MSVC 6.0 on the PC and with the system C++ compiler on UNIX.
4. 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

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
```

On PC:

Be sure to set your environment for MSVC++ 6.0. In order to do this, run

```
mex -setup
```

at the MATLAB command prompt. Then invoke MEX as:

```
mex mexcpp.cpp
```

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 [Technical Note 1601](#).
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's a MathWorks symbol, make sure you are linking against the correct libraries

Section 29: If loading fails

1. Make sure the MEX-file is a 32-bit DLL and doesn't use any 16-bit libraries
2. Make sure <MATLAB>/extern/lib/\$ARCH is included in \$LD_LIBRARY_PATH
3. Messages from ld.so usually indicate a problem with the library path
4. On the PC, make sure that the directories of all necessary external DLL's 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, please debug the MEX-file using the steps in the [debugging section of this technical note](#).

Please do not use the following form to request support.

Support assistance forms are available by clicking [contact us](#) on any of the support pages.

Did this information help? Yes No Didn't Apply

Is the level of technical detail appropriate? Yes Too Much Not Enough

What did you expect to find on this page that you want us to consider adding?

Additional Comments:

Submit Feedback

related topics:

[Demos](#) | [Search](#) | [Contact Support](#) | [Consulting](#) | [Press Room](#) | [Usability](#)

© 1994-2004 The MathWorks, Inc. [Trademarks](#) [Privacy Policy](#)